UNIVERSITY OF MALAYA

WID3006 / WIA1006 Machine Learning

Lab2 DL

Occ 2 Group ML

S2003637	Zhaodong Li
S2014548	Li Yang
S2000549	Yeyang Liu
S2000808	Fadjar Soengkono
S2011618	Ruojun Wang

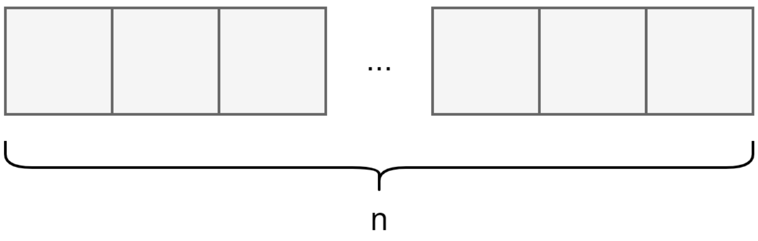
Part 1: TensorFlow

1. Explain the difference between a 1-D and a 2-D tensor

1-D tensor

- ❖ An 1-D tensor has one dimension (axes)
- ❖ It has a rank of 1
- ❖ It is in the form of a vector
- ❖ It is like a list of values

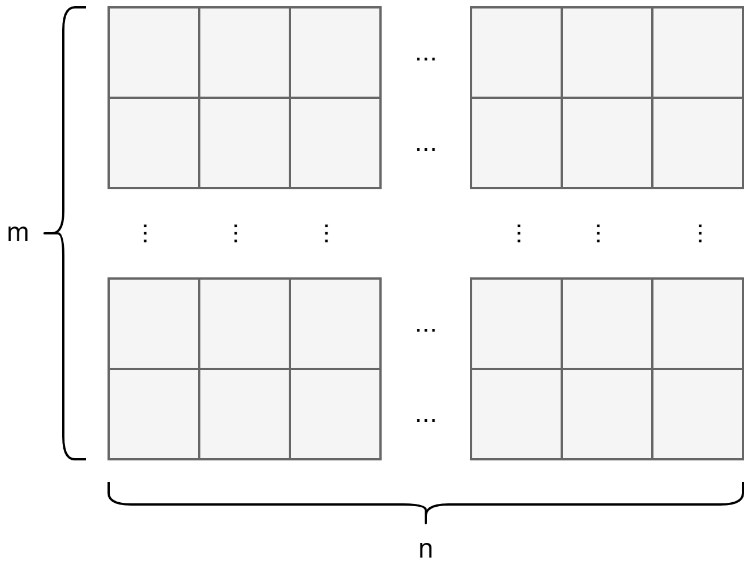
An 1-D tensor of shape [n]



2-D tensor

- ❖ A 2-D tensor has two dimensions (axes)
- ❖ It has a rank of 2
- ❖ It is in the form of a matrix
- ❖ It is like lists in a list

An 2-D tensor of shape [m, n]

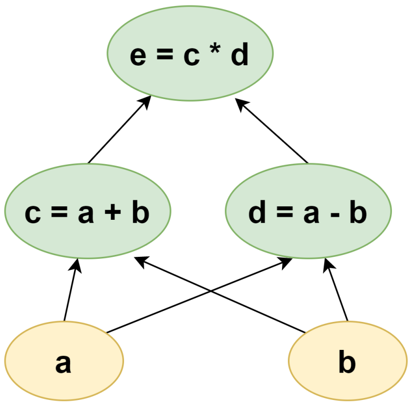


2. Provide the syntax to define a 4-D tensor with specific dimensions.

```python
d1 = 10
d2 = 150
d3 = 150
d4 = 1

images = tf.zeros([d1, d2, d3, d4])
```

3. Provide examples of defining tensor computations



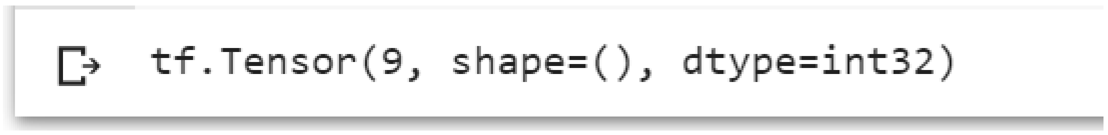
```python
def diff_of_square(a, b):
    # c = a + b
    c = tf.add(a, b)

    # d = a - b
    d = tf.subtract(a, b)

    # e = c * d
    e = tf.multiply(c, d)
    return e

if __name__ == "__main__":
    a, b = 5, 4
    e_out = diff_of_square(a, b)
    print(e_out)
```

output:



```
tf.Tensor(9, shape=(), dtype=int32)
```

4. What is Dense and what is Sequential in TensorFlow. Explain with the necessary examples.

Dense:

- Dense layer is a type of neural network layer, layers are the building block of neural networks.
- Dense refers to densely-connected (with its preceding layer). Dense layers are also referred to as fully connected layers. Neurons of the Dense layer are connected to every neuron of its preceding layer.
- Each Dense layer performs *output = activation(dot(input, kernel)* + *bias)*
- In tensorflow, the keras API provides a predefined Dense layer that can be customised specifying its arguments.
- To define a Dense layer, just simply create an instance of the Dense class, and pass the units and activation of the Dense layer you want. If the input shape is not given and the Dense layer is the first layer, input_shape must be specified.
- To better customise the arguments
    - *units* is the number of units in the Dense layer you want
    - *activation* is the activation function to be used in the Dense layer, if activation is not given, there will be no activation applied to this layer
    - *use_bias* is whether the layer uses a bias vector, the default value is True
    - *kernel_initializer* is the way of initialising the kernel, also referred to as weights, the default kernel initializer is 'glorot_uniform'
    - *bias_initializer* is the way of initialising the bias, the default bias initializer is 'zeros'
    - *kernel_regularizer* is the regularlizer applied to the kernel weights matrix, its default value is None
    - *bias_regularizer* is the regularizer function applied to the bias vector
    - *activity_regularizer* is the regularizer function applied to the output of the layer (its "activation")
    - *kernel_constraint* is the constraint function applied to the kernel weights matrix
    - *bias_constraint* is the constraint function applied to the bias vector

```python
# arguments that can be customised
layer = tf.keras.layers.Dense(
    units,
    activation=None,
    use_bias=True,
    kernel_initializer='glorot_uniform',
    bias_initializer='zeros',
    kernel_regularizer=None,
    bias_regularizer=None,
    activity_regularizer=None,
    kernel_constraint=None,
    bias_constraint=None,
    **kwargs
)

# example to define a simple dense layer
example_layer = tf.keras.layers.Dense(
    units = 32,
    activation = 'relu',
    input_shape = (16,)
)
```

Sequential

- tf.keras.Sequential provides an easy way to define a plain stack of layers into tf.keras.Model, in which each layer has exactly one input tensor and one output tensor
- Sequential also provides training and inference features on this model
- It is flexible to define a Sequential mode, you can define the model at a time by including all the layers, you can also add layers after you have defined the Sequential model.
- Sequential accepts two arguments
    - *layers* receives a list, in which the Layers to define the model are listed sequentially, its default value is None and does not have to have a value when an instance is created
    - *name* receives a string to be the name of the model, and will be displayed when you print the summary of the model

```
# arguments that can be used
tf.keras.Sequential(
    layers=None, name=None
)

# example to define a model using Sequential
model = tf.keras.models.Sequential([
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(units=120, activation='relu'),
    tf.keras.layers.Dense(units=84, activation='relu')
], name='example_layer')

# example to add a new layer
model.add(tf.keras.layers.Dense(units=10, activation='softmax'))
```

5. What is a SubClass in TensorFlow? How do you specify custom behavior using a SubClass?

What is a SubClass in Tensorflow

- keras API utilises object-oriented programming, a SubClass is a customizable child class that is inherited from the root class.
- Subclassing affords the flexibility to define custom layers, custom training loops, custom activation functions, and custom models.
- Model SubClassing is a fully customizable way to implement the feed-forward mechanism for our custom-designed deep neural network in an object-oriented fashion

How do you specify custom behavior using a SubClass?

- To specify custom behavior in Model SubClassing
  - Inherit your class from tf.keras.Model
  - in __init__ we define the Model's layers
  - in call define the network topology/graph inside the call function which is used to perform a forward-pass

```
class ResNet(tf.keras.Model):

    def __init__(self, num_classes=1000):
        super(ResNet, self).__init__()
        self.block_1 = ResNetBlock()
        self.block_2 = ResNetBlock()
        self.global_pool = layers.GlobalAveragePooling2D()
        self.classifier = Dense(num_classes)

    def call(self, inputs):
        x = self.block_1(inputs)
        x = self.block_2(x)
        x = self.global_pool(x)
        return self.classifier(x)
```

6. Explain how automatic differentiation works in TensorFlow

- Automatic differentiation is a set of techniques to evaluate the derivative of a function specified by a computer program. AD exploits the fact that every computer program, no matter how complicated, executes a sequence of elementary arithmetic operations. By applying the chain rule repeatedly to these operations, derivatives of arbitrary order can be computed automatically, accurately to working precision, and using at most a small constant factor more arithmetic operations than the original program.
- In tensorflow, when a forward pass is made through the network, all forward-pass operations get recorded to a "tape"; then, to compute the gradient, the tape is played backwards. By default, the tape is discarded after it is played backwards; this means that a particular tf.GradientTape can only compute one gradient, and subsequent calls throw a runtime error. However, we can compute multiple gradients over the same computation by creating a persistent gradient tape.

Part 2: Music Generation with RNNs

Provide a summary of how music generation is achieved with RNNs using TensorFlow.

Explain every step in detail.

Step 1: Preparation
1.1 Import dependencies.
1.2 Load dataset.
1.3 Explore the dataset
1.4 Find all unique characters in song strings

Step 2: Preprocessing
2.1 Vectorize the text
2.1.1 Create a numerical representation of our text-based dataset by generating two lookup tables

- one that maps characters to numbers
- a second that maps numbers back to characters

2.1.2 Vectorize the songs string using the char2idx table
2.2 Create training examples and targets
2.2.1 Divide the text into example sequences that we'll use during training. Each input sequence that we feed into our RNN will contain seq_length characters from the text
2.2.2 Define a target sequence for each input sequence, which will be used in training the RNN to predict the next character. For each input, the corresponding target will contain the same length of text, except shifted one character to the right

Step 3: Build RNN Model
3.1 Define a method to return the LSTM layer given rnn_units with custom values

- *rnn_units:* dimensionality of the output space
- *return_sequences=True*: return the last output
- *recurrent_initializer='glorot_uniform'*: use ‘glorot_uniform’ as initializer for the recurrent_kernel weights matrix, used for the linear transformation of the recurrent state
- *recurrent_activation='sigmoid'*: use ‘sigmoid’ as activation function to use for the recurrent step
- *stateful=True*: the last state for each sample at index i in a batch will be used as initial state for the sample of index i in the following batch

```
def LSTM(rnn_units):
  return tf.keras.layers.LSTM(
    rnn_units,
    return_sequences=True,
    recurrent_initializer='glorot_uniform',
    recurrent_activation='sigmoid',
    stateful=True,
  )
```

3.2 Build the RNN Model

3.2.1 Define method to build RNN Model using Sequential API

Layer 1: Embedding layer to transform indices into dense vectors of a fixed embedding size

Layer 2: LSTM layer with `rnn_units` number of units

Layer 3: Dense (fully-connected) layer that transforms the LSTM output into the vocabulary size

3.2.2 Use build_model method to build RNN Model with default hyperparameters

```
def build_model(vocab_size, embedding_dim, rnn_units, batch_size):
  model = tf.keras.Sequential([

    tf.keras.layers.Embedding(vocab_size, embedding_dim,
batch_input_shape=[batch_size, None]),

    LSTM(rnn_units),

    tf.keras.layers.Dense(vocab_size)
  ])

  return model

model = build_model(len(vocab), embedding_dim=256, rnn_units=1024,
batch_size=32)
```

Step 4: Test the RNN Model

4.1 Check the details of the model using *Model.summary*

4.2 Check the dimensionality of the output.

4.3 Take a look at predictions from the untrained model

Step 5: Train the RNN Model

5.1 Use tf.keras.losses.sparse_categorical_crossentropy as loss function to compute the loss

5.2 Define hyperparameters for training the model
5.3 Use hyperparameters to build the model
5.4 Use tf.keras.optimizers.Adam as optimizer and specify its learning rate
5.5 Define train_step
5.5.1 Use tf.GradientTape() to perform automatic differentiation
- Feed current input into the model and generate predictions
- Compute the loss

5.5.2 Use `model.trainable_variables` to get a list of all model parameters from gradient computation
5.5.3 Apply the gradients to the optimizer so it can update the model accordingly
5.5.4 Return the loss

```
@tf.function
def train_step(x, y):
  with tf.GradientTape() as tape:

    y_hat = model(x)

    loss = compute_loss(y, y_hat)

  grads = tape.gradient(loss, model.trainable_variables)


  optimizer.apply_gradients(zip(grads, model.trainable_variables))
  return loss
```

5.6 Train the model
5.6.1 Grab a batch and propagate it through the network
5.6.2 Update the progress bar
5.6.3 Update the model with the changed weights
5.6.4 Update the model with the changed weights every 100 iterations
5.6.5 Save the trained model and the weights

```
history = []

for iter in tqdm(range(num_training_iterations)):

  x_batch, y_batch = get_batch(vectorized_songs, seq_length,
batch_size)
  loss = train_step(x_batch, y_batch)

  history.append(loss.numpy().mean())
  plotter.plot(history)
  if iter % 100 == 0:
    model.save_weights(checkpoint_prefix)

model.save_weights(checkpoint_prefix)
```

Step 6: Generate music using the RNN Model

6.1 Restore the latest checkpoint

6.1.1 Rebuild the model using a batch_size=1 and the build_model method

6.1.2 Restore the model weights for the last checkpoint after training

6.1.3 Check the model using *Model.summary*

6.2 Generate prediction

- Initialise a "seed" start string and the RNN state, and set the number of characters we want to generate.
- Use the start string and the RNN state to obtain the probability distribution over the next predicted character.
- Sample from multinomial distribution to calculate the index of the predicted character. This predicted character is then used as the next input to the model.
- At each time step, the updated RNN state is fed back into the model, so that it now has more context in making the next prediction. After predicting the next character, the updated RNN states are again fed back into the model, which is how it learns sequence dependencies in the data, as it gets more information from the previous predictions.

6.2.1 Vectorize the starting string

6.2.2 Iterate over the generation_length times

- Evaluate the inputs and generate the next character predictions
- Remove the batch dimension
- Use a multinomial distribution to sample
- Pass the prediction along with the previous hidden state as the next inputs to the model
- Add the predicted character to the generated text

6.2.3 Return the generated string

6.3 Experiment and enjoy the generated songs!