Loop 结构实现 RNN

- 简单 ReLU RNN
- 静态单层单向 LSTM
- 静态单层双向 LSTM [TODO]
- dynamic shape 模式的单层单向 LSTM

简单 ReLU RNN

- 网络结构和输入输出数据与“RNNv2 层”保持一致，只是去掉了最高的一维

```
import numpy as np
from cuda import cudart
import tensorrt as trt

nBatchSize, nSequenceLength, nInputDim = 3, 4, 7  # 输入张量尺寸
nHiddenDim = 5  # 隐藏层宽度
data = np.ones([nBatchSize, nSequenceLength, nInputDim], dtype=np.float32)  # 输入数据
weightX = np.ones((nHiddenDim, nInputDim), dtype=np.float32)  # 权重矩阵 (X->H)
weightH = np.ones((nHiddenDim, nHiddenDim), dtype=np.float32)  # 权重矩阵 (H->H)
biasX = np.zeros(nHiddenDim, dtype=np.float32)  # 偏置 (X->H)
biasH = np.zeros(nHiddenDim, dtype=np.float32)  # 偏置 (H->H)

np.set_printoptions(precision=8, linewidth=200, suppress=True)
cudart.cudaDeviceSynchronize()

logger = trt.Logger(trt.Logger.ERROR)
builder = trt.Builder(logger)
network = builder.create_network(1 << int(trt.NetworkDefinitionCreationFlag.EXPLICIT_BATCH))
config = builder.create_builder_config()
config.max_workspace_size = 1 << 30
inputT0 = network.add_input('inputT0', trt.DataType.FLOAT, (nBatchSize, nSequenceLength, nInputDim))
#---------------------------------------------------------- --------------------# 替换部分
weightXLayer = network.add_constant([nInputDim, nHiddenDim], weightX.transpose().reshape(-1))
weightHLayer = network.add_constant([nHiddenDim, nHiddenDim], weightH.transpose().reshape(-1))
biasLayer = network.add_constant([nBatchSize, nHiddenDim], np.tile(biasX + biasH, (nBatchSize, 1)))
hidden0Layer = network.add_constant([nBatchSize, nHiddenDim], np.ones(nBatchSize * nHiddenDim,
dtype=np.float32))  # 初始隐藏状态，注意形状和 RNNV2层的不一样
lengthLayer = network.add_constant((), np.array([nSequenceLength], dtype=np.int32))  # 结果保留长度

loop = network.add_loop()
loop.add_trip_limit(lengthLayer.get_output(0), trt.TripLimit.COUNT)
iteratorLayer = loop.add_iterator(inputT0, 1, False)  # 每次抛出 inputTensor 的 H 维的一层
(nBatchSize,nInputDim)

rLayer = loop.add_recurrence(hidden0Layer.get_output(0))
_H0 = network.add_matrix_multiply(iteratorLayer.get_output(0), trt.MatrixOperation.NONE,
weightXLayer.get_output(0), trt.MatrixOperation.NONE)
_H1 = network.add_matrix_multiply(rLayer.get_output(0), trt.MatrixOperation.NONE,
weightHLayer.get_output(0), trt.MatrixOperation.NONE)
_H2 = network.add_elementwise(_H0.get_output(0), _H1.get_output(0), trt.ElementWiseOperation.SUM)
_H3 = network.add_elementwise(_H2.get_output(0), biasLayer.get_output(0), trt.ElementWiseOperation.SUM)
_H4 = network.add_activation(_H3.get_output(0), trt.ActivationType.RELU)
rLayer.set_input(1, _H4.get_output(0))
```

```
loopOutput0 = loop.add_loop_output(rLayer.get_output(0), trt.LoopOutput.LAST_VALUE, 0)  # 形状(nBatchSize,nHiddenDim)，nBatchSize 个独立输出，每个输出 1 个最终隐藏状态，每个隐藏状态 nHiddenDim 维坐标
loopOutput1 = loop.add_loop_output(_H4.get_output(0), trt.LoopOutput.CONCATENATE, 1)  # 形状(nSequenceLength,nBatchSize,nHiddenDim)，nBatchSize 个独立输出，每个输出 nSequenceLength 个隐藏状态，每个隐藏状态 nHiddenDim 维坐标
loopOutput1.set_input(1, lengthLayer.get_output(0))
#------------------------------------------------------- --------------------# 替换部分
network.mark_output(loopOutput0.get_output(0))
network.mark_output(loopOutput1.get_output(0))
engineString = builder.build_serialized_network(network, config)
engine = trt.Runtime(logger).deserialize_cuda_engine(engineString)
context = engine.create_execution_context()
_, stream = cudart.cudaStreamCreate()

inputH0 = np.ascontiguousarray(data.reshape(-1))
outputH0 = np.empty(context.get_binding_shape(1), dtype=trt.nptype(engine.get_binding_dtype(1)))
outputH1 = np.empty(context.get_binding_shape(2), dtype=trt.nptype(engine.get_binding_dtype(2)))
_, inputD0 = cudart.cudaMallocAsync(inputH0.nbytes, stream)
_, outputD0 = cudart.cudaMallocAsync(outputH0.nbytes, stream)
_, outputD1 = cudart.cudaMallocAsync(outputH1.nbytes, stream)

cudart.cudaMemcpyAsync(inputD0, inputH0.ctypes.data, inputH0.nbytes,
cudart.cudaMemcpyKind.cudaMemcpyHostToDevice, stream)
context.execute_async_v2([int(inputD0), int(outputD0), int(outputD1)], stream)
cudart.cudaMemcpyAsync(outputH0.ctypes.data, outputD0, outputH0.nbytes,
cudart.cudaMemcpyKind.cudaMemcpyDeviceToHost, stream)
cudart.cudaMemcpyAsync(outputH1.ctypes.data, outputD1, outputH1.nbytes,
cudart.cudaMemcpyKind.cudaMemcpyDeviceToHost, stream)
cudart.cudaStreamSynchronize(stream)

print("inputH0 :", data.shape)
print(data)
print("outputH0:", outputH0.shape)
print(outputH0)
print("outputH1:", outputH1.shape)
print(outputH1)

cudart.cudaStreamDestroy(stream)
cudart.cudaFree(inputD0)
cudart.cudaFree(outputD0)
cudart.cudaFree(outputD1)
```

- 输入张量形状 (3,4,7)，3 个独立输入，每个输入 4 个单词，每个单词 7 维坐标

$$\begin{bmatrix} \begin{bmatrix} 1. & 1. & 1. & 1. & 1. & 1. & 1. \\ 1. & 1. & 1. & 1. & 1. & 1. & 1. \\ 1. & 1. & 1. & 1. & 1. & 1. & 1. \\ 1. & 1. & 1. & 1. & 1. & 1. & 1. \end{bmatrix} \\ \begin{bmatrix} 1. & 1. & 1. & 1. & 1. & 1. & 1. \\ 1. & 1. & 1. & 1. & 1. & 1. & 1. \\ 1. & 1. & 1. & 1. & 1. & 1. & 1. \\ 1. & 1. & 1. & 1. & 1. & 1. & 1. \end{bmatrix} \\ \begin{bmatrix} 1. & 1. & 1. & 1. & 1. & 1. & 1. \\ 1. & 1. & 1. & 1. & 1. & 1. & 1. \\ 1. & 1. & 1. & 1. & 1. & 1. & 1. \\ 1. & 1. & 1. & 1. & 1. & 1. & 1. \end{bmatrix} \end{bmatrix}$$

- 输出张量 0 形状 (3,5)，3 个独立输出，每个隐藏状态 5 维坐标

$$\begin{bmatrix} 1717. & 1717. & 1717. & 1717. & 1717. \\ 1717. & 1717. & 1717. & 1717. & 1717. \\ 1717. & 1717. & 1717. & 1717. & 1717. \end{bmatrix}$$

- 输出张量 1 形状 (3,4,5)，3 个独立输出，每个包含 4 个隐藏状态，每个隐藏状态 5 维坐标

$$\begin{bmatrix} \begin{bmatrix} 12. & 12. & 12. & 12. & 12. \\ 67. & 12. & 12. & 12. & 12. \\ 342. & 342. & 342. & 342. & 342. \\ 1717. & 1717. & 1717. & 1717. & 1717. \end{bmatrix} \\ \begin{bmatrix} 12. & 12. & 12. & 12. & 12. \\ 67. & 67. & 67. & 67. & 67. \\ 342. & 342. & 342. & 342. & 342. \\ 1717. & 1717. & 1717. & 1717. & 1717. \end{bmatrix} \\ \begin{bmatrix} 12. & 12. & 12. & 12. & 12. \\ 67. & 67. & 67. & 67. & 67. \\ 342. & 342. & 342. & 342. & 342. \\ 1717. & 1717. & 1717. & 1717. & 1717. \end{bmatrix} \end{bmatrix}$$

- 计算过程：

$$\begin{aligned} h_1 &= \mathbf{ReLU}\left(W_{i,X}\cdot x_1 + W_{i,H}\cdot h_0 + b_{i,X} + b_{i,H}\right) \\ &= \mathbf{ReLU}\left(\begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{bmatrix} + \begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \right) \\ &= \mathbf{ReLU}\left((12,12,12,12,12)^{\mathrm{T}}\right) \\ &= (12,12,12,12,12)^{\mathrm{T}} \end{aligned}$$

$$\begin{aligned} h_2 &= \mathbf{ReLU}\left(W_{i,X}\cdot x_2 + W_{i,H}\cdot h_1 + b_{i,X} + b_{i,H}\right) \\ &= \mathbf{ReLU}\left(\begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{bmatrix} + \begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} 12 \\ 12 \\ 12 \\ 12 \\ 12 \\ 12 \\ 12 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \right) \\ &= \mathbf{ReLU}\left((67,67,67,67,67)^{\mathrm{T}}\right) \\ &= (67,67,67,67,67)^{\mathrm{T}} \end{aligned}$$

静态单层单向 LSTM

- 网络结构和输入输出数据与“RNNv2 层”保持一致，只是去掉了最高的一维
- 采用单输入网络，初始隐藏状态（h_0）和初始细胞状态（c_0）被写死成常量，需要改成变量的情况可以参考后面“dynamic shape 模式的单层单向 LSTM”部分

```
import numpy as np
from cuda import cudart
import tensorrt as trt

nBatchSize, nSequenceLength, nInputDim = 3, 4, 7  # 输入张量 NCHW
```

```
nHiddenDim = 5
x = np.ones([nBatchSize, nSequenceLength, nInputDim], dtype=np.float32)  # 输入数据
weightAllX = np.ones((nHiddenDim, nInputDim), dtype=np.float32)  # 权重矩阵 (X->H)
weightAllH = np.ones((nHiddenDim, nHiddenDim), dtype=np.float32)  # 权重矩阵 (H->H)
biasAllX = np.zeros(nHiddenDim, dtype=np.float32)  # 偏置 (X->H)
biasAllH = np.zeros(nHiddenDim, dtype=np.float32)  # 偏置 (H->H)

np.set_printoptions(precision=8, linewidth=200, suppress=True)
cudart.cudaDeviceSynchronize()

logger = trt.Logger(trt.Logger.ERROR)
builder = trt.Builder(logger)
network = builder.create_network(1 << int(trt.NetworkDefinitionCreationFlag.EXPLICIT_BATCH))
config = builder.create_builder_config()
config.max_workspace_size = 1 << 30
inputT0 = network.add_input('inputT0', trt.DataType.FLOAT, (nBatchSize, nSequenceLength, nInputDim))  #
采用单输入网络

#-------------------------------------------------------------------------------
def gate(network, x, wx, hiddenStateLayer, wh, b, isSigmoid):
    _h0 = network.add_matrix_multiply(x, trt.MatrixOperation.NONE, wx, trt.MatrixOperation.NONE)
    _h1 = network.add_matrix_multiply(hiddenStateLayer, trt.MatrixOperation.NONE, wh,
trt.MatrixOperation.NONE)
    _h2 = network.add_elementwise(_h0.get_output(0), _h1.get_output(0), trt.ElementWiseOperation.SUM)
    _h3 = network.add_elementwise(_h2.get_output(0), b, trt.ElementWiseOperation.SUM)
    _h4 = network.add_activation(_h3.get_output(0), [trt.ActivationType.TANH,
trt.ActivationType.SIGMOID][int(isSigmoid)])
    return _h4

weightAllXLayer = network.add_constant([nInputDim, nHiddenDim], weightAllX.transpose().reshape(-1))
weightAllHLayer = network.add_constant([nHiddenDim, nHiddenDim], weightAllH.transpose().reshape(-1))
biasAllLayer = network.add_constant([1, nHiddenDim], (biasAllX + biasAllH).reshape(-1))
hidden0Layer = network.add_constant([nBatchSize, nHiddenDim], np.ones(nBatchSize * nHiddenDim,
dtype=np.float32))  # 初始隐藏状态
cell0Layer = network.add_constant([nBatchSize, nHiddenDim], np.zeros(nBatchSize * nHiddenDim,
dtype=np.float32))  # 初始细胞状态
length = network.add_constant((), np.array([nSequenceLength], dtype=np.int32))

loop = network.add_loop()
loop.add_trip_limit(length.get_output(0), trt.TripLimit.COUNT)
iteratorLayer = loop.add_iterator(inputT0, 1, False)  # 每次抛出 inputTensor 的 H 维的一层
(nBatchSize,nInputDim), 双向 LSTM 要多一个反抛的迭代器
hiddenStateLayer = loop.add_recurrence(hidden0Layer.get_output(0))  # 一个 loop 中有多个循环变量
cellStateLayer = loop.add_recurrence(cell0Layer.get_output(0))

gateI = gate(network, iteratorLayer.get_output(0), weightAllXLayer.get_output(0),
hiddenStateLayer.get_output(0), weightAllHLayer.get_output(0), biasAllLayer.get_output(0), True)
gateC = gate(network, iteratorLayer.get_output(0), weightAllXLayer.get_output(0),
hiddenStateLayer.get_output(0), weightAllHLayer.get_output(0), biasAllLayer.get_output(0), False)
gateF = gate(network, iteratorLayer.get_output(0), weightAllXLayer.get_output(0),
hiddenStateLayer.get_output(0), weightAllHLayer.get_output(0), biasAllLayer.get_output(0), True)
gateO = gate(network, iteratorLayer.get_output(0), weightAllXLayer.get_output(0),
hiddenStateLayer.get_output(0), weightAllHLayer.get_output(0), biasAllLayer.get_output(0), True)

_h5 = network.add_elementwise(gateF.get_output(0), cellStateLayer.get_output(0),
trt.ElementWiseOperation.PROD)
_h6 = network.add_elementwise(gateI.get_output(0), gateC.get_output(0), trt.ElementWiseOperation.PROD)
newCellStateLayer = network.add_elementwise(_h5.get_output(0), _h6.get_output(0),
trt.ElementWiseOperation.SUM)
_h7 = network.add_activation(newCellStateLayer.get_output(0), trt.ActivationType.TANH)
```

```
newHiddenStateLayer = network.add_elementwise(gateO.get_output(0), _h7.get_output(0),
trt.ElementWiseOperation.PROD)

hiddenStateLayer.set_input(1, newHiddenStateLayer.get_output(0))
cellStateLayer.set_input(1, newCellStateLayer.get_output(0))

loopOutput0 = loop.add_loop_output(hiddenStateLayer.get_output(0), trt.LoopOutput.LAST_VALUE, 0)  # 形状
(nBatchSize,nHiddenDim)，nBatchSize 个独立输出，每个隐藏状态 nHiddenDim 维坐标
loopOutput1 = loop.add_loop_output(newHiddenStateLayer.get_output(0), trt.LoopOutput.CONCATENATE, 1)  #
形状 (nSequenceLength,nBatchSize,nHiddenDim)，nBatchSize 个独立输出，每个输出 nSequenceLength 个隐藏状态，每个
隐藏状态 nHiddenDim 维坐标
loopOutput1.set_input(1, length.get_output(0))
loopOutput2 = loop.add_loop_output(cellStateLayer.get_output(0), trt.LoopOutput.LAST_VALUE, 0)  # 形状
(nBatchSize,nHiddenDim)，nBatchSize 个独立输出，每个隐藏状态 nHiddenDim 维坐标
#---------------------------------------------------------------------------
network.mark_output(loopOutput0.get_output(0))
network.mark_output(loopOutput1.get_output(0))
network.mark_output(loopOutput2.get_output(0))
engineString = builder.build_serialized_network(network, config)
engine = trt.Runtime(logger).deserialize_cuda_engine(engineString)
context = engine.create_execution_context()
_, stream = cudart.cudaStreamCreate()

inputH0 = np.ascontiguousarray(x.reshape(-1))
outputH0 = np.empty(context.get_binding_shape(1), dtype=trt.nptype(engine.get_binding_dtype(1)))
outputH1 = np.empty(context.get_binding_shape(2), dtype=trt.nptype(engine.get_binding_dtype(2)))
outputH2 = np.empty(context.get_binding_shape(3), dtype=trt.nptype(engine.get_binding_dtype(3)))
_, inputD0 = cudart.cudaMallocAsync(inputH0.nbytes, stream)
_, outputD0 = cudart.cudaMallocAsync(outputH0.nbytes, stream)
_, outputD1 = cudart.cudaMallocAsync(outputH1.nbytes, stream)
_, outputD2 = cudart.cudaMallocAsync(outputH2.nbytes, stream)

cudart.cudaMemcpyAsync(inputD0, inputH0.ctypes.data, inputH0.nbytes,
cudart.cudaMemcpyKind.cudaMemcpyHostToDevice, stream)
context.execute_async_v2([int(inputD0), int(outputD0), int(outputD1), int(outputD2)], stream)
cudart.cudaMemcpyAsync(outputH0.ctypes.data, outputD0, outputH0.nbytes,
cudart.cudaMemcpyKind.cudaMemcpyDeviceToHost, stream)
cudart.cudaMemcpyAsync(outputH1.ctypes.data, outputD1, outputH1.nbytes,
cudart.cudaMemcpyKind.cudaMemcpyDeviceToHost, stream)
cudart.cudaMemcpyAsync(outputH2.ctypes.data, outputD2, outputH2.nbytes,
cudart.cudaMemcpyKind.cudaMemcpyDeviceToHost, stream)
cudart.cudaStreamSynchronize(stream)

print("x :", x.shape)
print(x)
print("outputH0:", outputH0.shape)
print(outputH0)
print("outputH1:", outputH1.shape)
print(outputH1)
print("outputH2:", outputH2.shape)
print(outputH2)

cudart.cudaStreamDestroy(stream)
cudart.cudaFree(inputD0)
cudart.cudaFree(outputD0)
cudart.cudaFree(outputD1)
cudart.cudaFree(outputD2)
```

- 输入张量形状 (3,4,7)，与“简单 ReLU RNN”的输入相同

- 输出张量 0 形状 (3,5)，为最终隐藏状态，3 个独立输出，每个隐藏状态 5 维坐标

$$\begin{bmatrix} 0.99932283 & 0.99932283 & 0.99932283 & 0.99932283 & 0.99932283 \\ 0.99932283 & 0.99932283 & 0.99932283 & 0.99932283 & 0.99932283 \\ 0.99932283 & 0.99932283 & 0.99932283 & 0.99932283 & 0.99932283 \end{bmatrix}$$

- 输出张量 1 形状 (3,4,5)，为所有隐藏状态，3 个独立输出，每个包含 4 个隐藏状态，每个隐藏状态 5 维坐标

$$\begin{bmatrix} \begin{bmatrix} 0.76158684 & 0.76158684 & 0.76158684 & 0.76158684 & 0.76158684 \\ 0.96400476 & 0.96400476 & 0.96400476 & 0.96400476 & 0.96400476 \\ 0.99504673 & 0.99504673 & 0.99504673 & 0.99504673 & 0.99504673 \\ 0.99932283 & 0.99932283 & 0.99932283 & 0.99932283 & 0.99932283 \end{bmatrix} \\ \begin{bmatrix} 0.76158684 & 0.76158684 & 0.76158684 & 0.76158684 & 0.76158684 \\ 0.96400476 & 0.96400476 & 0.96400476 & 0.96400476 & 0.96400476 \\ 0.99504673 & 0.99504673 & 0.99504673 & 0.99504673 & 0.99504673 \\ 0.99932283 & 0.99932283 & 0.99932283 & 0.99932283 & 0.99932283 \end{bmatrix} \\ \begin{bmatrix} 0.76158684 & 0.76158684 & 0.76158684 & 0.76158684 & 0.76158684 \\ 0.96400476 & 0.96400476 & 0.96400476 & 0.96400476 & 0.96400476 \\ 0.99504673 & 0.99504673 & 0.99504673 & 0.99504673 & 0.99504673 \\ 0.99932283 & 0.99932283 & 0.99932283 & 0.99932283 & 0.99932283 \end{bmatrix} \end{bmatrix}$$

- 输出张量 2 形状 (3,5)，为最终细胞状态，3 个独立输出，每个细胞状态 5 维坐标

$$\begin{bmatrix} 3.999906 & 3.999906 & 3.999906 & 3.999906 & 3.999906 \\ 3.999906 & 3.999906 & 3.999906 & 3.999906 & 3.999906 \\ 3.999906 & 3.999906 & 3.999906 & 3.999906 & 3.999906 \end{bmatrix}$$

- 计算过程：这里只用了一个 bias，$b_* = b_{*,X} + b_{?,H}$

$$\begin{aligned} I_1 = F_1 = O_1 &= \mathbf{sigmoid}\left(W_{*,X} \cdot x_1 + W_{*,H} \cdot h_0 + b_*\right) = (0.99999386, 0.99999386, 0.99999386, 0.99999386, 0.99999386)^{\mathrm{T}} \\ C_1 &= \mathbf{tanh}\left(W_{C,X} \cdot x_1 + W_{C,H} \cdot h_0 + b_C\right) = (0.99999999, 0.99999999, 0.99999999, 0.99999999, 0.99999999)^{\mathrm{T}} \\ c_1 &= F_1 \cdot c_0 + I_1 \cdot C_1 = (0.99999386, 0.99999386, 0.99999386, 0.99999386, 0.99999386)^{\mathrm{T}} \\ h_1 &= O_1 \cdot \mathbf{tanh}\left(c_1\right) = (0.76158690, 0.76158690, 0.76158690, 0.76158690, 0.76158690)^{\mathrm{T}} \end{aligned}$$

$$\begin{aligned} I_2 = F_2 = O_2 &= \mathbf{sigmoid}\left(W_{*,X} \cdot x_2 + W_{*,H} \cdot h_1 + b_*\right) = (0.99997976, 0.99997976, 0.99997976, 0.99997976, 0.99997976)^{\mathrm{T}} \\ C_2 &= \mathbf{tanh}\left(W_{C,X} \cdot x_2 + W_{C,H} \cdot h_1 + b_C\right) = (0.99999999, 0.99999999, 0.99999999, 0.99999999, 0.99999999)^{\mathrm{T}} \\ c_2 &= F_2 \cdot c_1 + I_2 \cdot C_2 = (1.99995338, 1.99995338, 1.99995338, 1.99995338, 1.99995338)^{\mathrm{T}} \\ h_2 &= O_2 \cdot \mathbf{tanh}\left(c_2\right) = (0.96400477, 0.96400477, 0.96400477, 0.96400477, 0.96400477)^{\mathrm{T}} \end{aligned}$$

静态单层双向 LSTM [TODO]

- 思路是使用两个迭代器，一个正抛一个反抛，最后把计算结果 concatenate 在一起

---

dynamic shape 模式的单层单向 LSTM

- 采用三输入网络，输入数据（x）、初始隐藏状态（h_0）和初始细胞状态（c_0）均独立输入

```
import numpy as np
from cuda import cudart
import tensorrt as trt

nBatchSize, nSequenceLength, nInputDim = 3, 4, 7  # 输入张量尺寸
nHiddenDim = 5
x = np.ones([nBatchSize, nSequenceLength, nInputDim], dtype=np.float32)  # 输入数据
h0 = np.ones([nBatchSize, nHiddenDim], dtype=np.float32)  # 初始隐藏状态
```

```
c0 = np.zeros([nBatchSize, nHiddenDim], dtype=np.float32)  # 初始细胞状态
weightAllX = np.ones((nHiddenDim, nInputDim), dtype=np.float32)  # 权重矩阵 (X->H)
weightAllH = np.ones((nHiddenDim, nHiddenDim), dtype=np.float32)  # 权重矩阵 (H->H)
biasAllX = np.zeros(nHiddenDim, dtype=np.float32)  # 偏置 (X->H)
biasAllH = np.zeros(nHiddenDim, dtype=np.float32)  # 偏置 (H->H)

np.set_printoptions(precision=8, linewidth=200, suppress=True)
cudart.cudaDeviceSynchronize()

logger = trt.Logger(trt.Logger.ERROR)
builder = trt.Builder(logger)
network = builder.create_network(1 << int(trt.NetworkDefinitionCreationFlag.EXPLICIT_BATCH))
profile = builder.create_optimization_profile()
config = builder.create_builder_config()
config.max_workspace_size = 4 << 30
inputT0 = network.add_input('inputT0', trt.DataType.FLOAT, (-1, -1, nInputDim))  # 3 输入，分别为 x，h0，c0
inputT1 = network.add_input('inputT1', trt.DataType.FLOAT, (-1, nHiddenDim))
inputT2 = network.add_input('inputT2', trt.DataType.FLOAT, (-1, nHiddenDim))
profile.set_shape(inputT0.name, (1, 1, nInputDim), (nBatchSize, nSequenceLength, nInputDim), (nBatchSize
* 2, nSequenceLength * 2, nInputDim))  # 范围覆盖住之后需要的值就好
profile.set_shape(inputT1.name, (1, nHiddenDim), (nBatchSize, nHiddenDim), (nBatchSize * 2, nHiddenDim))
profile.set_shape(inputT2.name, (1, nHiddenDim), (nBatchSize, nHiddenDim), (nBatchSize * 2, nHiddenDim))
config.add_optimization_profile(profile)

#-----------------------------------------------------------------------------
def gate(network, xTensor, wx, hTensor, wh, b, isSigmoid):
    _h0 = network.add_matrix_multiply(xTensor, trt.MatrixOperation.NONE, wx, trt.MatrixOperation.NONE)
    _h1 = network.add_matrix_multiply(hTensor, trt.MatrixOperation.NONE, wh, trt.MatrixOperation.NONE)
    _h2 = network.add_elementwise(_h0.get_output(0), _h1.get_output(0), trt.ElementWiseOperation.SUM)
    _h3 = network.add_elementwise(_h2.get_output(0), b, trt.ElementWiseOperation.SUM)
    _h4 = network.add_activation(_h3.get_output(0), [trt.ActivationType.TANH,
trt.ActivationType.SIGMOID][int(isSigmoid)])
    return _h4

weightAllXLayer = network.add_constant([nInputDim, nHiddenDim],
np.ascontiguousarray(weightAllX.transpose().reshape(-1)))
weightAllHLayer = network.add_constant([nHiddenDim, nHiddenDim],
np.ascontiguousarray(weightAllH.transpose().reshape(-1)))
biasAllLayer = network.add_constant([1, nHiddenDim], np.ascontiguousarray(biasAllX + biasAllH))

_t0 = network.add_shape(inputT0)
_t1 = network.add_slice(_t0.get_output(0), [1], [1], [1])
_t2 = network.add_shuffle(_t1.get_output(0))  # 两个循环条件都需要标量输入
_t2.reshape_dims = ()

loop = network.add_loop()
loop.add_trip_limit(_t2.get_output(0), trt.TripLimit.COUNT)
iteratorLayer = loop.add_iterator(inputT0, 1, False)  # 每次抛出 inputT0 的一层 (nBatchSize,nInputDim)，双
向 LSTM 要多一个反抛的迭代器

hiddenStateLayer = loop.add_recurrence(inputT1)  # 初始隐藏状态和细胞状态，一个 loop 中有多个循环变量
cellStateLayer = loop.add_recurrence(inputT2)

gateI = gate(network, iteratorLayer.get_output(0), weightAllXLayer.get_output(0),
hiddenStateLayer.get_output(0), weightAllHLayer.get_output(0), biasAllLayer.get_output(0), True)
gateF = gate(network, iteratorLayer.get_output(0), weightAllXLayer.get_output(0),
hiddenStateLayer.get_output(0), weightAllHLayer.get_output(0), biasAllLayer.get_output(0), True)
gateC = gate(network, iteratorLayer.get_output(0), weightAllXLayer.get_output(0),
hiddenStateLayer.get_output(0), weightAllHLayer.get_output(0), biasAllLayer.get_output(0), False)
```

```
gateO = gate(network, iteratorLayer.get_output(0), weightAllXLayer.get_output(0),
hiddenStateLayer.get_output(0), weightAllHLayer.get_output(0), biasAllLayer.get_output(0), True)

_h5 = network.add_elementwise(gateF.get_output(0), cellStateLayer.get_output(0),
trt.ElementWiseOperation.PROD)
_h6 = network.add_elementwise(gateI.get_output(0), gateC.get_output(0), trt.ElementWiseOperation.PROD)
newCellStateLayer = network.add_elementwise(_h5.get_output(0), _h6.get_output(0),
trt.ElementWiseOperation.SUM)
_h7 = network.add_activation(newCellStateLayer.get_output(0), trt.ActivationType.TANH)
newHiddenStateLayer = network.add_elementwise(gateO.get_output(0), _h7.get_output(0),
trt.ElementWiseOperation.PROD)

hiddenStateLayer.set_input(1, newHiddenStateLayer.get_output(0))
cellStateLayer.set_input(1, newCellStateLayer.get_output(0))

loopOutput0 = loop.add_loop_output(hiddenStateLayer.get_output(0), trt.LoopOutput.LAST_VALUE, 0)  # 形状
(nBatchSize,nHiddenSize), nBatchSize 个独立输出，每个隐藏状态 nHiddenSize 维坐标
loopOutput1 = loop.add_loop_output(newHiddenStateLayer.get_output(0), trt.LoopOutput.CONCATENATE, 1)  #
形状 (nBatchSize,nSequenceLength,nHiddenSize)，nBatchSize 个独立输出，每个输出 nSequenceLength 个隐藏状态，每个
隐藏状态 nHiddenSize 维坐标
loopOutput1.set_input(1, _t2.get_output(0))
loopOutput2 = loop.add_loop_output(cellStateLayer.get_output(0), trt.LoopOutput.LAST_VALUE, 0)  # 形状
(nBatchSize,nHiddenSize)，nBatchSize 个独立输出，每个隐藏状态 nHiddenSize 维坐标
#------------------------------------------------------------------------------
network.mark_output(loopOutput0.get_output(0))
network.mark_output(loopOutput1.get_output(0))
network.mark_output(loopOutput2.get_output(0))
engineString = builder.build_serialized_network(network, config)
engine = trt.Runtime(logger).deserialize_cuda_engine(engineString)
context = engine.create_execution_context()
context.set_binding_shape(0, [nBatchSize, nSequenceLength, nInputDim])
context.set_binding_shape(1, [nBatchSize, nHiddenDim])
context.set_binding_shape(2, [nBatchSize, nHiddenDim])
_, stream = cudart.cudaStreamCreate()

para = np.load('para_tf_keras_layers_LSTM.npz')
inputH0 = np.ascontiguousarray(x.reshape(-1))
inputH1 = np.ascontiguousarray(h0.reshape(-1))
inputH2 = np.ascontiguousarray(c0.reshape(-1))
outputH0 = np.empty(context.get_binding_shape(3), dtype=trt.nptype(engine.get_binding_dtype(3)))
outputH1 = np.empty(context.get_binding_shape(4), dtype=trt.nptype(engine.get_binding_dtype(4)))
outputH2 = np.empty(context.get_binding_shape(5), dtype=trt.nptype(engine.get_binding_dtype(5)))
_, inputD0 = cudart.cudaMallocAsync(inputH0.nbytes, stream)
_, inputD1 = cudart.cudaMallocAsync(inputH1.nbytes, stream)
_, inputD2 = cudart.cudaMallocAsync(inputH2.nbytes, stream)
_, outputD0 = cudart.cudaMallocAsync(outputH0.nbytes, stream)
_, outputD1 = cudart.cudaMallocAsync(outputH1.nbytes, stream)
_, outputD2 = cudart.cudaMallocAsync(outputH2.nbytes, stream)

cudart.cudaMemcpyAsync(inputD0, inputH0.ctypes.data, inputH0.nbytes,
cudart.cudaMemcpyKind.cudaMemcpyHostToDevice, stream)
cudart.cudaMemcpyAsync(inputD1, inputH1.ctypes.data, inputH1.nbytes,
cudart.cudaMemcpyKind.cudaMemcpyHostToDevice, stream)
cudart.cudaMemcpyAsync(inputD2, inputH2.ctypes.data, inputH2.nbytes,
cudart.cudaMemcpyKind.cudaMemcpyHostToDevice, stream)
context.execute_async_v2([int(inputD0), int(inputD1), int(inputD2), int(outputD0), int(outputD1),
int(outputD2)], stream)

cudart.cudaMemcpyAsync(outputH0.ctypes.data, outputD0, outputH0.nbytes,
cudart.cudaMemcpyKind.cudaMemcpyDeviceToHost, stream)
```

```
cudart.cudaMemcpyAsync(outputH1.ctypes.data, outputD1, outputH1.nbytes,
cudart.cudaMemcpyKind.cudaMemcpyDeviceToHost, stream)
cudart.cudaMemcpyAsync(outputH2.ctypes.data, outputD2, outputH2.nbytes,
cudart.cudaMemcpyKind.cudaMemcpyDeviceToHost, stream)
cudart.cudaStreamSynchronize(stream)

print("x :", x.shape)
print(x)
print("h0:", h0.shape)
print(h0)
print("c0:", c0.shape)
print(c0)
print("outputH0:", outputH0.shape)
print(outputH0)
print("outputH1:", outputH1.shape)
print(outputH1)
print("outputH2:", outputH2.shape)
print(outputH2)

cudart.cudaStreamDestroy(stream)
cudart.cudaFree(inputD0)
cudart.cudaFree(inputD1)
cudart.cudaFree(inputD2)
cudart.cudaFree(outputD0)
cudart.cudaFree(outputD1)
cudart.cudaFree(outputD2)
```

- 输入输出张量形状和结果与“静态单层单向 LSTM”完全相同