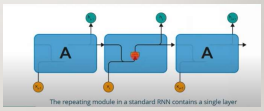
LONG SHORT TERM MEMORY NETWORKS

LSTM

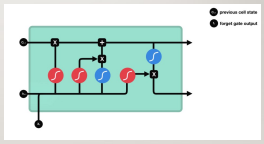
- Long Short Term Memory Networks- usually just called "LSTMs" are a special kind of RNN
- They are capable of Learning long-term dependencies.



LSTM

- **Forget Gate**

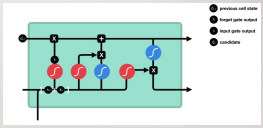
First, we have the forget gate. This gate decides what information should be thrown away or kept. Information from the previous **hidden state** and information from the **current input** is passed through the sigmoid function.



LSTM

- **Input Gate**

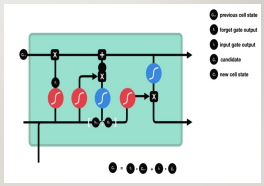
To update the cell state, we have the input gate. First, we pass the previous hidden state and current input into a sigmoid function. That decides which values will be updated. Then **hidden state** and **current input** into the tanh function. Then you **multiply the tanh output with the sigmoid output**



LSTM

- **Cell State**

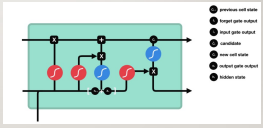
Now we should have enough information to calculate the cell state. First, the cell state gets **pointwise multiplied by the forget vector.** Then we take the **output from the input gate and do a pointwise addition** which updates the cell state to new values. That gives us our new cell state.



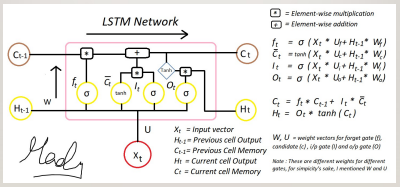
LSTM

- Output Gate

First, we pass the **previous hidden state and the current input** into a sigmoid function. Then we pass the newly modified cell state to the tanh function. We multiply the tanh output with the sigmoid output to decide what information the hidden state should carry. The output is the hidden state. The new cell state and the new hidden is then carried over to the next time step.



LSTM



CODE

```
# Imports
import torch
import torchvision  # torch package for vision related things
import torch.nn.functional as F  # Parameterless functions, like (some) activation functions
import torchvision.datasets as datasets  # Standard datasets
import torchvision.transforms as transforms  # Transformations we can perform on our dataset for augmentation
from torch import optim  # For optimizers like SGD, Adam
from torch import nn  # All neural network modules
from torch.utils.data import DataLoader  # Gives easier dataset managment by creating mini batches etc.
from tqdm import tqdm  # For a visual progress bar!
import numpy as np
import matplotlib.pyplot as plt
# Set device ( if GPU is available all parameters copy into GPU and start execution otherwise CPU)
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
```

CODE

```
# Hyperparameters (A hyperparameter is that is set before the learning process begins. These parameters affect how well a model trains)
#each image is size of (1x 28 x 28) (EXPECTED FEATURES)
input_size = 28
sequence_length = 28
#Number of nodes in hidden layers
hidden_size = 256
#number of Recurrent Layers in models
num_layers = 2
#Mnist dataset have 10 classes (hand written digits from 0 to 9)
num_classes = 10
# a tuning parameter in an optimization algorithm that determines the step size at each iteration while moving toward a minimum of a loss function.
learning_rate = 0.005
#The batch size defines the number of samples that will be propagated through the network.
batch_size = 64
#the number times that the learning algorithm will work through the entire training dataset. One epoch means that each sample in the training dataset has had an opportunity to update the internal model parameters
num_epochs = 2
```

CODE

```
# Recurrent neural network with LSTM (many-to-one)
class RNN_LSTM(nn.Module): #nn.module is a parent class (inherited) call by RNN which is child class
    def __init__(self, input_size, hidden_size, num_layers, num_classes):
        super(RNN_LSTM, self).__init__() #give access to child class in parent class
        self.hidden_size = hidden_size #give number of nodes in hidden layer to model
        self.num_layers = num_layers  #Number of Recurrent layers
        #input all hyperparameter in the model..... IF batch_first TRUE then the input and output tensors are provided as (batch, seq, feature)
        self.lstm = nn.LSTM(input_size, hidden_size, num_layers, batch_first=True)
        # "hidden_size * sequence_length" is number of input features and num_classes is output features.linear transformation to the incoming data: y = xA^T + b
        self.fc = nn.Linear(hidden_size * sequence_length, num_classes) #After linear transformation output will fully connected
```

CODE

```
def forward(self, x): # input data x
        # Set initial hidden and cell states
        #creat H0(initial state output) and c0(initial cell memory)  tesnor with scaler vlue 0 of (x.size(0) give number of rows so its define the shape of tensor)
        h0 = torch.zeros(self.num_layers, x.size(0), self.hidden_size).to(device)
        c0 = torch.zeros(self.num_layers, x.size(0), self.hidden_size).to(device)
        # Forward propagate LSTM
        out, _ = self.lstm(x, (h0, c0))
        # out: tensor of shape (batch_size, seq_length, hidden_size) we find  dimension of out to figure it outby giving(-1) and count its number of rows with shape(0).
        out = out.reshape(out.shape[0], -1)
        # Decode the hidden state of the last time step
        out = self.fc(out) #fully connected layer
        return out
```

CODE

```
# download data from tenserflow
train_dataset = datasets.MNIST(root="dataset/", train=True, transform=transforms.ToTensor(), download=True)
test_dataset = datasets.MNIST(root="dataset/", train=False, transform=transforms.ToTensor(), download=True)
#load the data with batches given in batch_size and shuffle the data
train_loader = DataLoader(dataset=train_dataset, batch_size=batch_size, shuffle=True)
test_loader = DataLoader(dataset=test_dataset, batch_size=batch_size, shuffle=True)

# Initialize Recurrent neural network
model = RNN(input_size, hidden_size, num_layers, num_classes).to(device)

# Loss function and optimizer
#CrossEntropy is used as loss function
criterion = nn.CrossEntropyLoss()
#use Adam optimzer with model parameters and LR
optimizer = optim.Adam(model.parameters(), lr=learning_rate)
```

CODE

```
# Train Network
for epoch in range(num_epochs):
    # enumerate data with batch id (data is input images and targets are labels) and tqdm is used for visual progress bar
    for batch_idx, (data, targets) in enumerate(tqdm(train_loader)):
        # set data to cuda if possible
        # squeeze(1) if input is of shape: (A×1×B×C×1×D) then the out tensor will be of shape: (A×B×C×D).
        data = data.to(device=device).squeeze(1) #input images and check gpu is availbale or not
        targets = targets.to(device=device) #labels and check gpu is availbale or not
        # forward
        #calculate the model ouput(predictions)
        scores = model(data)
        #now loss function is calculate with model presictions and actual output
        loss = criterion(scores, targets)
        #for every mini-batch during the training phase, we typically want to explicitly set the gradients to zero before
starting to do backpropragation because PyTorch accumulates the gradients on subsequent backward passes
        #if it not zero the gradient would be a combination of the old gradient, which you have already used to update your
model parameters, and the newly-computed gradient.
        optimizer.zero_grad()
        loss.backward()
        # gradient descent update step/adam step
        optimizer.step()
```

CODE

```
# Check accuracy on training & test to see how good our model
def check_accuracy(loader, model):
    num_correct = 0
    num_samples = 0
    # Set model to eval
    model.eval() # model in evaluating mode so deactivates all dropout layers or training is stop
    with torch.no_grad(): #stop gradient calculation
        #x is input and Y is output (data and labels)
        for x, y in loader: #load contain data x is input image and y is label data
            # squeeze remove single-dimensional entries
            x = x.to(device=device).squeeze(1)
            y = y.to(device=device)
            #calculate scores
            scores = model(x)
```

CODE

```
        _, predictions = scores.max(1)
            # y contain actual label output it compare with prediction to calculate total number of number
    correct predictions
            num_correct += (predictions == y).sum()
            # Number of all predictions .size(0) give number of rows
            num_samples += predictions.size(0)
    # Toggle model back to train
    model.train()
    return num_correct / num_samples
#print the accuracy
print(f"Accuracy on training set: {check_accuracy(train_loader, model)*100:2f}")
print(f"Accuracy on test set: {check_accuracy(test_loader, model)*100:.2f}")
```

CODE

```
# Disable grad
with torch.no_grad():#stop gradient calculation
    # Retrieve item
    index = 256
    item = test_dataset[index]
    image = item[0] # take first image
    true_target = item[1] # take actual output
    # Generate prediction
    prediction = model(image)
    # Predicted class value using argmax.....(returns indices of the max element of the array in a particular axis)
    predicted_class = np.argmax(prediction)
    # Reshape image
    image = image.reshape(28,28, 1)
    # Show result
    plt.imshow(image,cmap='gray')
    plt.title(f"Prediction:{predicted_class}--Actual target:{true_target}")
    plt.show()
```

REFRENCES

- https://www.edureka.co/
- https://towardsdatascience.com/

THANKS