HyperionDev

Mastering Sequential Models

Task

Visit our website

Introduction

Recurrent Neural Networks (RNNs) are powerful tools for processing sequential data. However, they suffer from the vanishing gradient problem, making it difficult to learn long-term dependencies. Long Short-Term Memory (LSTM) and Gated Recurrent Unit (GRU) networks address this issue by introducing memory cells and gates that regulate the flow of information. In this lesson, we'll delve into the limitations of RNNs, explore the mechanisms of LSTMs and GRUs, and implement these models using PyTorch to tackle challenging sequential tasks.

Vanishing and Exploding Gradients

While it's possible to unfold the network for each training example, this approach significantly increases computational complexity, which would be proportional to $T \times W$, where T is the number of unfolding steps and W is the number of training words. However, if we only unfold and train the recurrent part after several training examples, the complexity can be reduced. In fact, unfolding after all training examples would make the complexity depend solely on W. An example of how gradients flow in batch mode training of an RNN is shown in Figure 1.

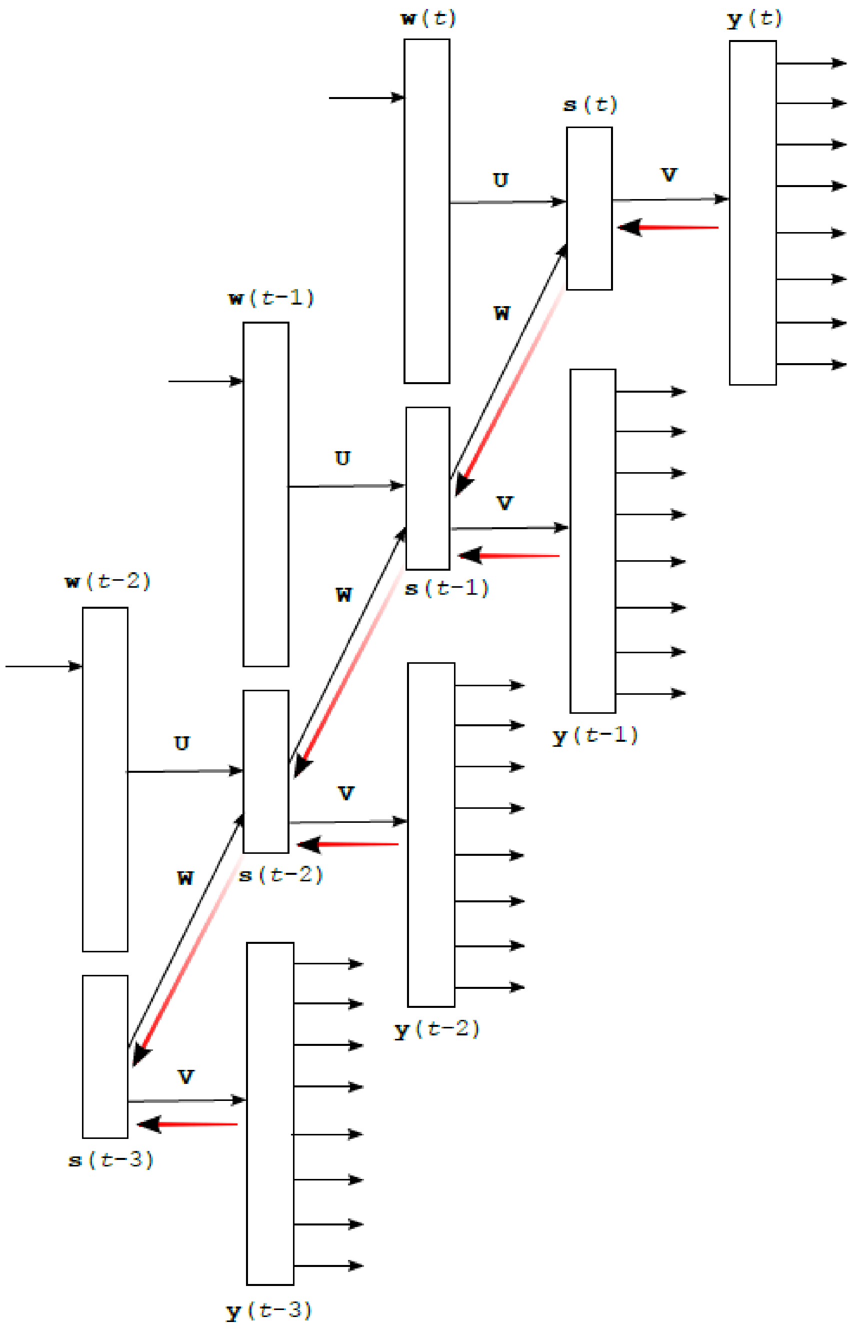

Figure 1: Illustration of batch mode training in an RNN. Red arrows show the propagation of gradients through the network as it is unfolded. (Mikolov, 2012).

In Figure 1, we see the structure of a simple RNN unfolded over four-time steps ($t-3$, $t-2$, $t-1$, t). The network consists of input vectors $\omega(t)$, $\omega(t-1)$, $\omega(t-2)$, and at each time step, which are processed sequentially. The hidden states $s(t)$, $s(t-1)$, $s(t-2)$, and $s(t-3)$ are calculated at each step based on the input vectors and the hidden state from the previous step. The weight matrices U, W, and V are used to compute the hidden states and the outputs $y(t)$, $y(t-1)$, $y(t-2)$, and $y(t-3)$. The red arrows in the figure represent the propagation of gradients through the network during backpropagation, showing how the error gradients are passed backward through time steps to update the weights. When an error is detected at the output layer, this error is backpropagated through each time step to update the weights $U, W,$ and V. This process allows the network to adjust the weights based on the entire sequence of inputs, not just the current input, enabling the RNN to learn long-term dependencies. However, it is important to highlight that this backpropagation through multiple time steps can lead to very small gradients (vanishing gradients) or very large gradients (exploding gradients), both of which can hinder the training process.

For numerical stability, using double precision for real numbers and adding some form of regularisation is advisable. Using single precision without regularisation may lead to convergence issues, resulting in suboptimal solutions. Additionally, training RNNs can be more challenging than training normal feedforward networks due to the potential for exploding gradients. In some cases, the gradients can increase to such large values during backpropagation through the recurrent connections that they overwrite the network weights with meaningless values, causing training failure.

Problem description

A key issue in training RNNs is learning long-term dependencies, known as the vanishing gradient problem. This occurs when gradients of the loss function, used to update network weights, either shrink to zero (vanish) or grow too large (explode) as they are propagated backward through time during training. This challenge has been articulated in research (Bengio et al., 1994).

When training RNNs, the backpropagation through time (BPTT) algorithm unfolds the network across each timestep of input data, treating the sequence as layers of a deep network. The output at each timestep depends on the hidden state from the previous timestep, formulated as:

$$h_t = f\left(W \cdot h_{t-1} + U \cdot x_t + b\right),$$

where h_t is the hidden state at time t, x_t is the input at time t, W and U are weight matrices, and b is a bias term. The function f typically is a non-linear activation function like sigmoid or tanh.

The gradients of a loss function L with respect to the weights W are computed by the chain rule across these timesteps, which involves terms like the partial derivative of the hidden state at time t with respect to the hidden state at time $t - 1$, $\frac{\partial h_t}{\partial h_{t-1}}$. If the sequence is lengthy or if the derivatives of the activation functions are small, this repeated multiplication can cause the gradient values to exponentially approach zero as they propagate back through time, resulting in slow or stalled training. Conversely, if the derivatives are large, the gradients can exponentially grow as they are backpropagated, resulting in substantial updates to the weights and causing the model to diverge.

The gradient ∇_t of the loss at timestep t with respect to the weights can be recursively described as:

$$\nabla_t = \frac{\partial L}{\partial h_t} \prod_{k=t}^{1} \frac{\partial h_k}{\partial h_{k-1}}.$$

This product of derivatives can decay to zero (vanish) or explode if the values are large enough, as seen in RNNs dynamics.

These phenomena make it challenging to train RNNs to capture dependencies over long sequences, as required in applications like language translation, where the meaning of a sentence can depend on words much earlier in the sequence. In practice, this means either using techniques to manage these gradients, such as gradient clipping (to combat exploding gradients) or employing alternative RNN architectures like Long Short-Term Memory (LSTM) networks and Gated Recurrent Units (GRU), which are designed to mitigate the vanishing gradient problem by introducing gating mechanisms to control the flow of information.

Impact on training RNNs

The impact of vanishing and exploding gradients on the training of RNNs will be analysed next, while it directly influences the efficiency and effectiveness of these models, especially in learning tasks involving long sequences.

When gradients vanish during training, the weights associated with long-term dependencies receive very small updates because the gradient terms become infinitesimally small. This problem arises primarily due to the nature of the algorithm used in RNNs. Mathematically, the updates to the weights in an RNN depend on the gradients computed as follows:

$$\Delta W = \eta \sum_{t=1}^{T} \frac{\partial L}{\partial W_t},$$

where η is the learning rate, T is the total number of timesteps, L is the loss function, and W_t represents the weight at time t. The partial derivative of the loss with respect to the weights at each timestep, $\frac{\partial L}{\partial W_t}$, involves terms from earlier timesteps, propagated forward through the network's recurrent connections:

$$\frac{\partial L}{\partial W_t} = \frac{\partial L}{\partial h_t} \frac{\partial h_t}{\partial W_t}.$$

The recursive nature of h_t $\left(h_t = f\left(W \cdot h_{t-1} + U \cdot x_t + b\right)\right)$ means that computing $\frac{\partial h_t}{\partial W_t}$ involves products of derivatives stretching back to the initial timesteps. If the derivatives are small (common with sigmoid or tanh activation functions), their products over many timesteps approach zero. As a result, the gradient contributions from early timesteps become negligible, leading to a lack of meaningful weight updates for parts of the network that handle earlier data in the sequence. This obstructs the network's ability to learn dependencies between distant events in a sequence, which is damaging in tasks such as language modelling or any temporal pattern recognition where context from the distant past is informative.

Conversely, exploding gradients occur when the products of derivatives lead to excessively large updates to weights. This usually happens when the derivatives are large, which can be the case with certain activation functions or initialization schemes. The mathematical concern here is similar to that of vanishing gradients but in the opposite direction:

$$\frac{\partial L}{\partial W_t} = \frac{\partial L}{\partial h_t} \frac{\partial h_t}{\partial W_t}$$

If $\frac{\partial h_t}{\partial h_{t-1}}$ is large, the repeated multiplication over many timesteps can cause the gradient to grow exponentially as the BPTT progresses from $t = T$ to $t = 1$. This exponential growth in gradient magnitude can lead to weight updates that are too large, causing the learning process to diverge instead of converging to a minimum of the loss function. This makes training unstable and often necessitates the use of gradient clipping, a technique where gradients exceeding a certain threshold are scaled down before being used to update weights, to prevent divergence.

To address these challenges, various techniques have been employed. Gradient clipping, which caps the magnitude of gradients during backpropagation, helps prevent the vanishing and exploding gradient problems. Additionally, LSTM and GRU networks incorporate sophisticated gating mechanisms that regulate the flow of information, enabling them to capture long-term dependencies and maintain stable gradients over extended sequences compared to traditional RNNs.

Long Short-Term Memory (LSTM)

Long Short-Term Memory (LSTM) neural networks represent a special type of RNN designed to overcome challenges related to learning long-range dependencies in sequence data. As we saw previously, the standard RNNs struggle with these because of issues like vanishing and exploding gradients. LSTMs tackle this problem by incorporating a complex mechanism of gates that control the flow of information.

Architecture and components

An LSTM unit typically consists of a cell (which holds the state of the network over time), an input gate, an output gate, and a forget gate (Figure 2).

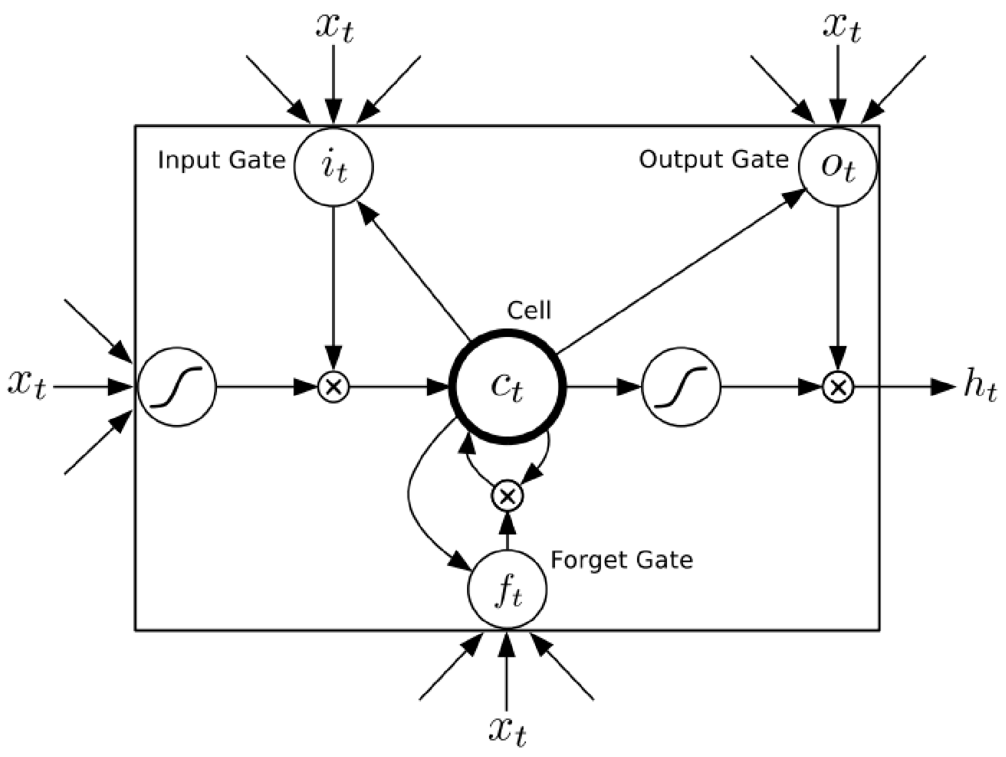

Figure 2: LSTM Cell (Graves, 2014).

These components work together to regulate the information flow in and out of the cell, making it possible for the network to maintain a long-term memory. The equations governing these processes are as follows:

- **Forget gate**

$$f_t = \sigma\left(W_f \bullet \left[h_{t-1}, x_t\right] + b_f\right)$$

The forget gate determines which information from the previous cell state C_{t-1} should be forgotten. It uses the previous hidden state h_{t-1} and the current input x_t, combined and passed through a sigmoid function σ. The sigmoid function outputs values between 0 and 1, where values close to 0 mean "forget" and values close to 1 mean "retain."

- **Input gate**

$$i_t = \sigma\left(W_i \bullet \left[h_{t-1}, x_t\right] + b_i\right)$$

$$\tilde{C}_t = \tanh\left(W_C \bullet \left[h_{t-1}, x_t\right] + b_C\right)$$

  The input gate regulates which new information should be added to the cell state. It uses a sigmoid function to decide which values to update. A tanh function creates a vector of new candidate values that could be added to the state.

- **Cell state update**

$$C_t = f_t \bullet C_{t-1} + i_t \bullet \tilde{C}_t$$

  The cell state is updated by combining the previous cell state C_{t-1} (scaled by the forget gate f_t) and the new candidate values $\tilde{C}_t$ (scaled by the input gate i_t). This allows the LSTM to retain important information over long periods.

- **Output gate**

$$o_t = \sigma\left(W_o \bullet \left[h_{t-1}, x_t\right] + b_o\right)$$

$$h_t = o_t \bullet \tanh\left(C_t\right)$$

  Finally, the output gate decides what the next hidden state should be h_t. It uses the current cell state C_t, which is filtered through a tanh function to ensure the values are between -1 and 1. The output is then scaled by the output gate o_t, which decides which parts of the cell state will be output as the hidden state.

LSTMs are used in situations where the context or state over time is an important factor, such as in language modelling, speech recognition, and time series prediction. Their ability to capture long-term dependencies makes them superior to standard RNNs for tasks involving sequences with complex structures, improving the performance on a variety of tasks in natural language processing and other areas of machine learning.

Addressing vanishing and exploding gradients

Addressing the problem of vanishing and exploding gradients represents an important challenge for training deep neural networks, especially RNNs like LSTM networks. These challenges impact the network's ability to learn, as they affect the stability and speed of the convergence during training. Below, we analyse how these issues are addressed, focusing on LSTM networks.

As was previously discussed, exploding gradients occur when the gradients calculated during the training process become too large, often leading to numerical instability and divergent behaviour in the learning process. This results in model parameters updating with excessively large values, which can cause the model to overshoot optimal points during training.

A common technique to mitigate exploding gradients is **gradient clipping** that involves setting a threshold value. If the norm of the gradient exceeds this threshold, it is scaled down to keep its norm at or below the threshold. Mathematically, this can be expressed as:

$$\text{if } \|\nabla\theta\| > v, \text{ then } \nabla\theta = \frac{v \cdot \nabla\theta}{\|\nabla\theta\|},$$

where $\nabla\theta$ represents the gradient of the model parameters, and v is the threshold. This operation ensures that the updates to the parameters do not cause the training process to become unstable.

As for vanishing gradients, they occur when the gradients become too small, effectively preventing the weights from changing their values, which makes learning stall. This is problematic in deep networks with many layers, as the gradient signal can decay exponentially during backpropagation through the layers. The architecture of LSTMs inherently addresses the vanishing gradient problem through its use of gates and cell states. These components allow it to maintain a stable gradient across time steps as follows:

- **Forget gate:** This gate helps control the extent to which a value remains in the cell state, thus allowing the network to drop values that are no longer relevant to the prediction task.
- **Input and output gates:** These regulate the flow of input signals into the cell state and output signals into the subsequent layers, thus controlling the propagation of gradients during backpropagation.
- **Cell state updates:** The cell state is designed to mitigate the vanishing gradient by linearly combining the old state (scaled by the forget gate) with the new candidate values (scaled by the input gate), which provides a way for gradients to be calculated that does not involve repetitive multiplication by potentially small numbers.

Extra resource

In 2006, Marcus Hutter, Jim Bowery, and Matt Mahoney initiated a challenge known as the **Hutter Prize**. The goal was to compress the initial 100 million bytes of the English Wikipedia into the smallest possible file. This file had to encompass both the compressed data and the code responsible for the compression algorithm.

This process not only tested the efficiency of compression algorithms but also provided a unique dataset for training neural networks, allowing them to learn from a diverse range of characters and structures beyond standard text corpora, highlighting the capability of LSTM networks to handle complex, multilayered data sequences over long ranges.

Gated Recurrent Units (GRUs)

GRUs are a streamlined variant of LSTM cells designed to handle similar issues with training over long sequences, such as the vanishing gradient problem. GRUs were introduced in 2014 by K. Cho et al. as a modification of RNN. Despite these simplifications, GRUs have demonstrated performance comparable to LSTMs in various applications such as polyphonic music modelling, speech signal modelling, and natural language processing.

GRUs in Practice

First, let’s explore GRUs in real-world situations. For instance, GRUs are frequently employed in polyphonic music modelling to predict note sequences within a musical piece. This task necessitates the model's ability to learn long-term dependencies as the prediction of a note may depend on notes that occurred significantly earlier in the sequence. Researchers have been able to faithfully record these dependencies and create cohesive music sequences by employing GRUs. For instance, Kulshrestha (2020) presented how Google's Magenta project models polyphonic music using GRUs, and produces compositions in classical composers' styles.

GRUs are also used in speech signal modelling, where the aim is to generate or recognise spoken language. In such tasks, managing long-term dependencies is crucial as the pronunciation of a word can be influenced by following words. By effectively capturing context across extended sequences, GRUs contribute to the improvement of speech recognition systems. The LibriSpeech dataset, containing thousands of hours of English speech, is commonly used to train and evaluate these models.

In natural language processing, GRUs are widely used for tasks such as sentiment analysis, text generation, and machine translation. In machine translation, for instance, GRUs can handle longer phrases by maintaining context across multiple words, leading to more accurate translations. A commonly used dataset for training these models is the WMT14 dataset, which comprises parallel texts in various languages. Leveraging GRUs, Google's Neural Machine Translation (GNMT) system has achieved state-of-the-art performance in numerous language pairs (Ding, 2017).

GRU architecture

GRUs modify the LSTM design by combining the input and forget gates into a single update gate and by removing the output gate entirely. This results in a model with fewer parameters than LSTM, as it simplifies the gating mechanism without significantly compromising the network's ability to capture dependencies in sequence data. This simplified gating in GRUs is not only computationally more efficient but also helps in quicker convergence during training.

GRU gates

A GRU has two main gates: the update gate z_t and the reset gate r_t. These gates are calculated using the following formulas:

$$z_t = \sigma\left(W_z x_t + U_z h_{t-1} + b_z\right)$$

$$r_t = \sigma\left(W_r x_t + U_r h_{t-1} + b_r\right)$$

Here, σ represents the sigmoid function, W and U are weight matrices, b is a bias term, x_t is the input at time t, h_{t-1} is the hidden state from the previous time step.

The candidate hidden state, $\tilde{h}_t$, which is analogous to the cell input in LSTMs, is computed as follows:

$$\tilde{h}_t = \tanh\left(W_h x_t + U_h\left(r_t \odot h_{t-1}\right) + b_h\right)$$

The operator $\odot$ denotes the **Hadamard product**. This equation introduces the reset gate, which determines how much of the previous hidden state should influence the new candidate state, allowing the model to forget irrelevant parts of the previous state if necessary.

The final step in computing the current hidden state h_t uses the update gate to blend the previous hidden state and the new candidate hidden state:

$$h_t = (1 - z_t) \odot \tilde{h}_t + z_t \odot h_{t-1}$$

This blending mechanism allows the GRU to carry information across many time steps without removing the effects of short-term inputs, thereby addressing the vanishing gradient problem effectively.

Alternative activation functions can also be used, provided they guarantee outputs in the range of [0, 1]. New activation functions can be created by modifying the update gate(z_t) and reset gate (r_t) in a GRU architecture (Dey & Salem, 2017):

- Type 1, each gate depends only on the previous hidden state and the bias.

$$z_t = \sigma(U_z h_{t-1} + b_z)$$

$$r_t = \sigma(U_r h_{t-1} + b_r)$$

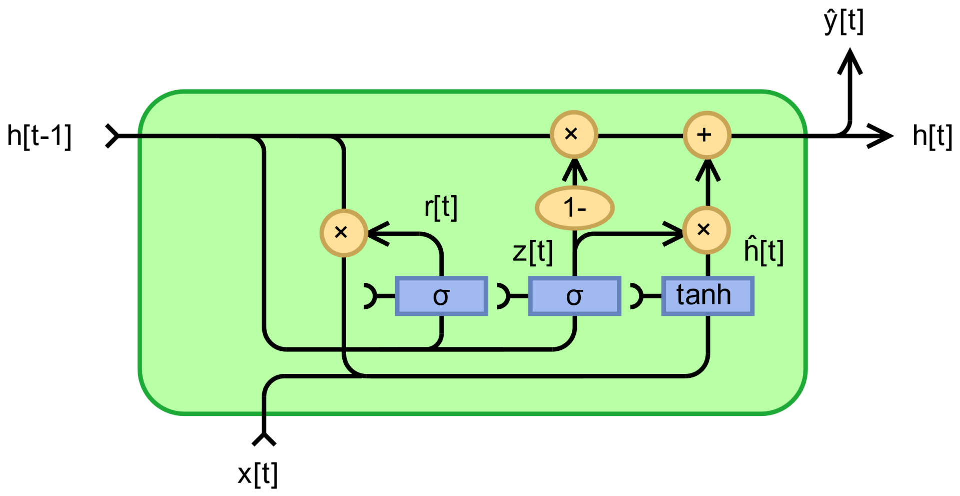

Figure 3: Type 1 (Jeblad, 2018a) CC BY-SA 4.0

The reset gate r_t influences the candidate hidden state $\tilde{h}_t$ by controlling how much of the previous hidden state h_{t-1} is considered. The update gate z_t determines the final hidden state h_t by blending the previous hidden state h_{t-1} and the candidate hidden state $\tilde{h}_t$.

- Type 2, each gate depends only on the previous hidden state.

$$z_t = \sigma\left(U_z h_{t-1}\right)$$

$$r_t = \sigma\left(U_r h_{t-1}\right)$$

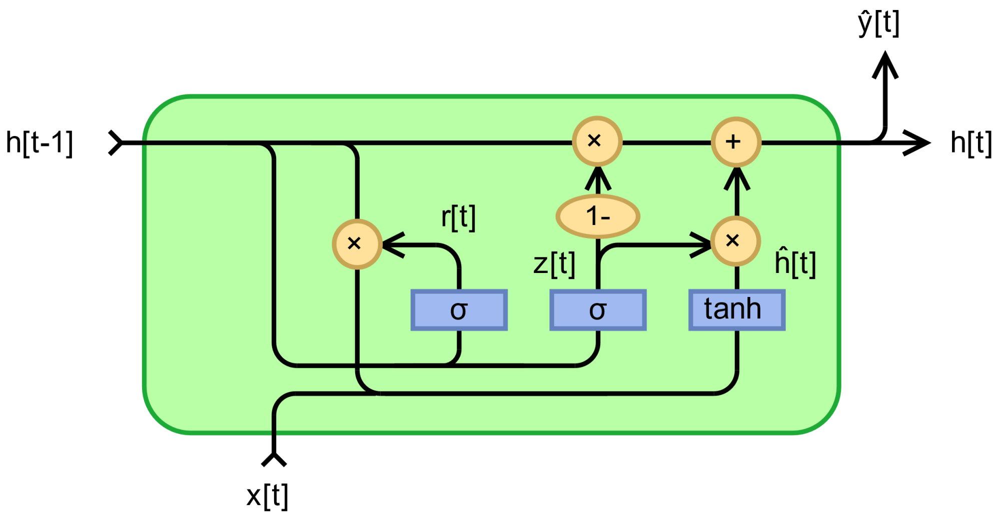

Figure 4: Type 2 (Jeblad, 2018b) CC BY-SA 4.0

- Type 3, each gate is computed using only the bias.

$$z_t = \sigma\left(b_z\right)$$

$$r_t = \sigma\left(b_r\right)$$

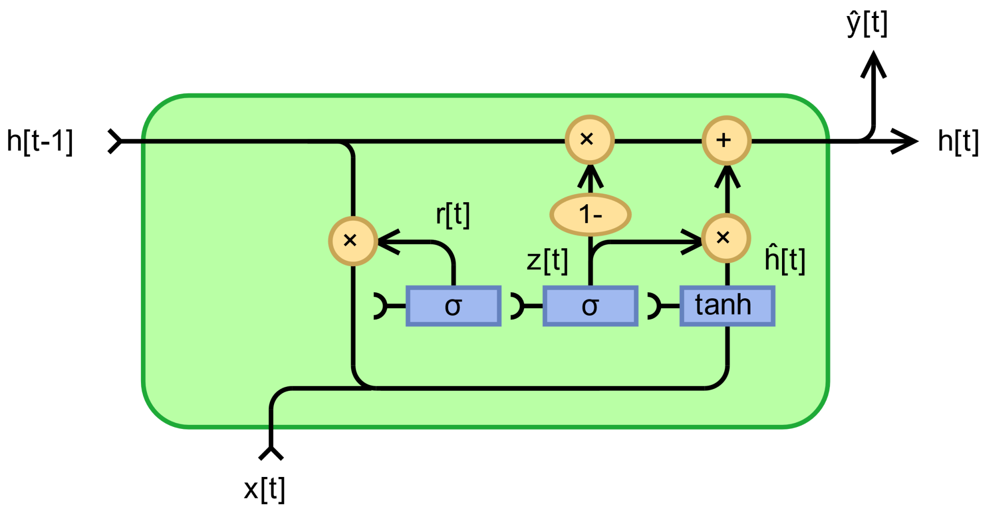

Figure 5: Type 3 (Jeblad, 2018c) CC BY-SA 4.0.

Practical Implementations

In this section, we will explain how to implement an LSTM and GRU model with PyTorch. We'll be working with a dataset which contains daily temperature recordings for a city. Our task will be to predict the daily minimum temperatures by exploiting a trained LSTM or GRU model.

LSTM example using PyTorch

Importing libraries and dataset

As usual, the code at the top of your file should import the libraries you will require. In addition to `numpy`, `pandas` and `sklearn` you will need PyTorch and its modules (`torch`, `torch.nn`, `torch.optim`) to build the LSTM model.

```
import numpy as np
import pandas as pd
import torch
import torch.nn as nn
import torch.optim as optim
from sklearn.preprocessing import MinMaxScaler
from sklearn.metrics import mean_squared_error
```

Ensure reproducibility by setting a random seed. We need to set a random seed for both NumPy and PyTorch.

```
# Set the random seed
np.random.seed(40)
torch.manual_seed(40)
```

Data Processing

Next, let's import our dataset and process it.

```
# Load the dataset
data_url =
"https://raw.githubusercontent.com/jbrownlee/Datasets/master/daily-min-temper
atures.csv"
df = pd.read_csv(data_url)
```

Note: If you have issues accessing the dataset via GitHub the csv file is provided in your folder.

The following code snippet will extract the values from the "Temp" column of a DataFrame, convert them into floating-point numbers, and reshape them into a two-dimensional array with a single column.

```
temperatures = (
    df.reset_index(drop=True)["Temp"].values.astype(float).reshape(-1, 1))
```

Once we have extracted the required values, the next step is to normalise the **`temperatures`** data using Min-Max scaling, transforming it to a range between 0 and 1.

```
scaler = MinMaxScaler(feature_range=(0, 1))
data_normalized = scaler.fit_transform(temperatures)
```

Next, we will create sequences of input-output pairs from the given dataset. It takes two parameters: **`data`**, which is the input dataset, and **`seq_length`**, which specifies the length of each sequence. It iterates through the data and creates input sequences of length **`seq_length`** (X) and their corresponding output values (y). Each input sequence is formed by taking **`seq_length`** consecutive elements from the data, and the corresponding output value is the element immediately following the sequence. Finally, it returns the input-output pairs as NumPy arrays.

```
def create_sequences(data, seq_length):
    """ Generates input-output pairs from the provided data."""
    X, y = [], []
    for i in range(len(data) - seq_length):
        X.append(data[i:i+seq_length])
        y.append(data[i+seq_length])
    return np.array(X), np.array(y)
```

Now, let's manually define the number of time steps to look back.

```
seq_length = 20
X, y = create_sequences(data_normalized, seq_length)
```

It is time to split the data into training and testing sets.

```
train_size = int(len(X) * 0.67)
X_train, X_test = X[:train_size], X[train_size:]
y_train, y_test = y[:train_size], y[train_size:]
```

Before training the model we need to convert the data into PyTorch tensor format. Tensors are similar to NumPy ndarrays but are more computationally efficient because they can run on GPUs.

```
X_train_tensor = torch.tensor(X_train, dtype=torch.float32)
y_train_tensor = torch.tensor(y_train, dtype=torch.float32)
X_test_tensor = torch.tensor(X_test, dtype=torch.float32)
y_test_tensor = torch.tensor(y_test, dtype=torch.float32)
```

Initialising an LSTM

It is time to initialise our LSTM model. The following code snippet defines an LSTM-based neural network model using PyTorch, where the class `LSTMModel` inherits from `nn.Module`. In its constructor, it initialises an LSTM layer (`self.lstm`) with the specified input size, hidden size and number of layers. It also defines a fully connected layer (`self.fc`) to map the LSTM output to the desired output size. In the forward method, it takes an input tensor `x`, passes it through the LSTM layer, extracts the output of the last time step, and feeds it to the fully connected layer to produce the final output.

```
class LSTMModel(nn.Module):
    """ Defines an LSTM-based neural network model with one LSTM layer and
        one fully connected layer for sequence prediction tasks. """
    def __init__(self, input_size, hidden_size, output_size, num_layers):
        super(LSTMModel, self).__init__()
        self.lstm = nn.LSTM(input_size, hidden_size, num_layers,
batch_first=True)
        self.fc = nn.Linear(hidden_size, output_size)

    def forward(self, x):
        output, _ = self.lstm(x)
        output = self.fc(output[:, -1, :])
        return output
```

Once we have defined our class, it is very easy to set the hyperparameters for our model: the number of features in the input data (`input_size`), the number of units in the LSTM hidden layer (`hidden_size`), the dimensionality of the model's output (`output_size`), the number of hidden layers (`num_layers`), the rate at which the model adjusts its parameters during training (`learning_rate`), and the number of times the entire dataset is passed forward and backwards through the model during training (`num_epochs`). Finally, we are specifying the L2 regularisation parameter (`l2_penalty`) as a required step for introducing the L2 regularisation technique. We want to introduce L2 regularisation because it helps prevent overfitting by penalising large weights, which encourages the model to keep weights small.

Except for the `input_size` and `output_size` parameters, which should remain unchanged, all other hyperparameters can be adjusted as needed. We encourage you to experiment with these values through repeated training to achieve better prediction performance with the model.

```python
# Define hyperparameters for the model training
input_size = 1
hidden_size = 50  # Number of features in the hidden state
output_size = 1
num_layers = 2  # Number of LSTM layers
learning_rate = 0.001  # Learning rate
num_epochs = 100
l2_penalty = 0.01  # L2 regularisation parameter
```

After specifying the hyperparameters, we can Initialise the LSTM model, loss function, and optimizer. Here, an LSTM model is instantiated with specified input, hidden and output sizes, as well as the number of layers. The mean squared error loss function (`MSELoss`) is chosen as the criterion for training, and the `Adam` optimiser is employed with a given learning rate and L2 penalty as a weight decay factor to update the model parameters during optimisation.

```python
model = LSTMModel(input_size, hidden_size, output_size, num_layers)
criterion = nn.MSELoss()
optimizer = optim.Adam(model.parameters(), lr=learning_rate,
weight_decay=l2_penalty)
```

Training an LSTM

Now, it is time to train the model. The loop in the next code snippet trains the LSTM model over a set number of epochs. Within each epoch, gradients are reset before passing the input data through the model to generate predictions. The mean squared error (MSE) loss between the predictions and actual values is calculated. Backpropagation is then used to compute gradients, which are utilised to update the model parameters through optimisation using the Adam optimizer. The loss for the current epoch is printed to monitor training progress.

```python
# Training loop for the LSTM model
for epoch in range(num_epochs):
    optimizer.zero_grad()
    output = model(X_train_tensor)
    loss = criterion(output, y_train_tensor)
    loss.backward()
    optimizer.step()
    print(f'Epoch [{epoch+1}/{num_epochs}], Loss: {loss.item():.4f}')
```

According to the printed loss values below (to save space here, we showed the first five and the last five printed rows), it could be concluded that the decreasing trend in the loss values over the epochs suggests that the model is effectively learning from the training data. This reduction indicates that the model is becoming more accurate in predicting the target values as training progresses. The consistency in the decrease demonstrates that the model is converging towards an optimal solution, with diminishing improvements in performance as training continues.

Output:

```
Epoch [1/100], Loss: 0.3191
Epoch [2/100], Loss: 0.3060
Epoch [3/100], Loss: 0.2931
Epoch [4/100], Loss: 0.2803
Epoch [5/100], Loss: 0.2675
.
.
.
Epoch [95/100], Loss: 0.0166
Epoch [96/100], Loss: 0.0165
Epoch [97/100], Loss: 0.0165
Epoch [98/100], Loss: 0.0164
Epoch [99/100], Loss: 0.0164
Epoch [100/100], Loss: 0.0163
```

Predicting and Evaluating

At this point, we could make some predictions for the daily minimum temperatures (in degrees Celsius) using the trained model with the test data and print them. However, before printing, to present the output values appropriately, we should inversely transform the predicted values back to their original scale using a scaler.

```
# Predicting with the model using test data and inverting the scaling of
predictions
with torch.no_grad():
    y_pred = model(X_test_tensor)

y_pred_inv = scaler.inverse_transform(y_pred.numpy())

print("Predicted values:", y_pred_inv.squeeze())
```

Output:

```
Predicted values: [10.604856 10.598566 10.559222 ... 10.725738 10.727158
10.744257
```

Ok, we have the predicted values of the daily minimum temperatures, but let's quantify the model's performance using the MSE, a popular metric for evaluating regression tasks.

```
mse = mean_squared_error(y_test, y_pred_inv.squeeze())
print("Mean Squared Error:", mse)
```

Output:

```
Mean Squared Error: 104.29889310930383
```

An MSE of 104.29 indicates certain discrepancies between predicted and actual values. This suggests further refinement of the model is needed to improve accuracy. Try to independently change the offered values of hyperparameters of the model offered above, training instructions and the regularisation technique in order to improve the performance of the model.

LSTM example using Keras

The PyTorch implementation provides a clear and hands-on understanding of how an LSTM model is built and trained. It requires a lot of manual handling, which can be advantageous in research or situations where you want to fine tune every aspect of the model.

In the next section, we will explore how we can use a higher-level API called **Keras** to build the same LSTM. Keras provides a more concise way to build, train, and evaluate neural networks. It allows us to focus on the model architecture rather than the mechanics of training.

The code for data loading, preprocessing and creating the training and testing datasets is the same for both PyTorch and Keras. The only thing we will need to change from the beginning is how we set the seed. Keras is a higher-level API which runs on top of TensorFlow, so the Tensorflow library will be used to set the seed along with Numpy. Since TensorFlow is required, it must be installed before proceeding

Before installing anything, ensure that you set up a **virtual environment** to prevent conflicts with other Python packages. Once your environment is activated, install TensorFlow by running the following command in your terminal or command prompt:

```
pip install tensorflow
```

Ensure you have a compatible Python version installed, as TensorFlow supports versions 3.8 to 3.11. To download one of these versions, visit the **official Python website** and install the appropriate version. During installation, make sure to select the option to add Python to the system path.

```
# Set random seed for reproducibility
```

```
import tensorflow as tf

np.random.seed(40)

tf.random.set_seed(40)
```

That is the only amendment we will make on the pre-training code. From here, we will maintain the same code that yielded the model inputs for the PyTorch example, and go straight into defining and training the model using Keras.

Importing Necessary Libraries

Because Keras is a different library altogether, we will need to import some additional libraries ontop of some of the ones we used for the PyTorch example.

```
import numpy as np

import pandas as pd

from sklearn.preprocessing import MinMaxScaler

from sklearn.metrics import mean_squared_error

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import LSTM, Dense

from tensorflow.keras.optimizers import Adam

from tensorflow.keras.regularizers import l2
```

Initialising and Building the LSTM

Unlike in the PyTorch case, we do not need to define a LSTMModel class. Instead, we will use Sequential which sets up a linear model where layers are added in order, providing a simple architecture which is ideal for adding and stacking layers.

```
model = Sequential()
```

We will make use of the same hyperparameters as before.

```
# Define hyperparameters for the model training
input_size = 1
hidden_size = 50  # Number of features in the hidden state
output_size = 1
num_layers = 2  # Number of LSTM layers
learning_rate = 0.001  # Learning rate
num_epochs = 100
l2_penalty = 0.01  # L2 regularisation parameter
```

Below, we add a single LSTM layer with 50 units added, processing the sequential input of 20 time steps. We set return_sequences=False so that only the last time step's output is used. A Dense layer is added to produce the final prediction.

```
model.add(LSTM(hidden_size, return_sequences=False, input_shape=(seq_length,

         input_size), kernel_regularizer=l2(l2_penalty)))

model.add(Dense(output_size))
```

We then compile the model, preparing it for training.

```
model.compile(optimizer=Adam(learning_rate=learning_rate), loss="mse")
```

Training the Model

Now it is time to train the model. The fit() function trains the LSTM model over **100 epochs** using the training dataset. A **batch size of 32** means the model processes **32 sequences at a time** before updating weights. The verbose parameter is set so the loss value is displayed to monitor training progress.

```
model.fit(X_train, y_train, epochs=num_epochs, batch_size=32, verbose=1)
```

Predicting and Evaluating

The model training is complete, so we can predict values on the test dataset to evaluate its performance. Just as before, we will rescale the predictions using inverse_transform. Finally, we evaluate the performance of the model.

```
y_pred = model.predict(X_test)

y_pred_inv = scaler.inverse_transform(y_pred)

mse = mean_squared_error(scaler.inverse_transform(y_test),

      y_pred_inv.squeeze())

print("Mean Squared Error:", mse)
```

Key Differences Between PyTorch & Keras

Feature	PyTorch	Keras
Model Definition	Class based using nn.Module	Sequential API
Training Loop	Manual for loop	model.fit() handles the training loop
Optimiser Setup	Use optim.Adam()	Defined inside compile()
Predictions	With torch.no_grad()	model.predict()

GRU example

The practical implementation of a GRU network is very similar to the implementation of an LSTM network. Refer to the previous example and practical implementation of our LSTM model. Our modifications will be done in two simple steps.

First, you need to change the code behind building the class. The main difference between initialising the GRU and LSTM classes is in the recurrent layer used within the GRU model. The GRU model utilises an `nn.GRU` layer, and in the forward method, the `GRUModel` class calls the GRU layer on the input data.

```
class GRUModel(nn.Module):
    def __init__(self, input_size, hidden_size, output_size, num_layers=1):
        super(GRUModel, self).__init__()
        self.gru = nn.GRU(input_size, hidden_size, num_layers,
batch_first=True)
        self.fc = nn.Linear(hidden_size, output_size)

    def forward(self, x):
        output, _ = self.gru(x)
        output = self.fc(output[:, -1, :])
        return output
```

The second difference is in changing the existing line of initialising the model with the following one:

```
model = GRUModel(input_size, hidden_size, output_size, num_layers)
```

That's it. Run the code, train the model, and evaluate its performance. Compare the results between the LSTM and GRU models. Which model performs better in predicting the daily minimum temperatures?

Instructions

The practical implementations discussed above are provided in your task folder. Read through these examples and make sure you understand each step before tackling the practical tasks.

Practical task 1

In this task, you will build and train an LSTM model to forecast housing prices using the **California Housing dataset**.

1. Use the **sequential_task.ipynb** notebook in your folder to complete this task.

- The libraries you need have been imported for you including the sklearn datasets package to access the `california_housing` dataset.

2. Set the random seed for reproducibility to 50, to ensure that the random number generation is consistent across runs.
3. The California Housing dataset is loaded for you. Use the **sklearn documentation** to answer the following questions in a Markdown cell:
    - What is a `Bunch`?
    - What is the default type for `data`?
    - What is the default type for the `target`?
4. Normalise the features and target values using `MinMaxScaler`.
5. Next, use the `create_sequence` function to generate sequences of data and targets. This function should also be used to prepare the dataset with a specified sequence length.
6. Split the dataset into training (67%) and testing (33%) sets and convert the training and testing data from NumPy arrays to PyTorch tensors.
7. Implement an LSTM model class with configurable input size, hidden size, output size, and number of layers.
8. Define hyperparameters, including hidden size, number of layers, learning rate, and L2 regularization.
9. Instantiate the LSTM model, define the loss function, and set up the optimizer with L2 regularization.
10. Implement a training loop to train the model for a specified number of epochs.
11. Print the loss for each epoch to monitor training progress.
12. Use the trained model to make predictions on the test set. Don't forget to invert the scaling of the predictions to the original range.
13. Finally, calculate and print the MSE between the true and predicted values.

Practical task 2

1. Your second practical task is to build a model again to forecast housing prices using the California Housing dataset, but this time implement a GRU model. You may use the same **sequential_task.ipynb** Jupyter notebook.

2. Exclude the regularisation process from both models and answer these questions in a Markdown cell.
    a. Has performance gotten worse in both models?
    b. What is the importance of regularisation for optimising the efficiency of models?
3. Include the regularisation process again. Now, try to improve your LSTM and GRU models by adjusting the hyperparameters. Try at least two different configurations for each model. Did any of your models perform better after your adjustments? Summarise the changes you made and the performance for each set of hyperparameters in a Markdown cell.

Important: Be sure to upload all files required for the task submission inside your task folder and then click "Request review" on your dashboard.

Share your thoughts

Please take some time to complete this short feedback **form** to help us ensure we provide you with the best possible learning experience.

Reference list

Bengio, Y., Simard, P., & Frasconi, P. (1994). Learning long-term dependencies with gradient descent is difficult. IEEE transactions on neural networks, 5(2), 157-166.

Cho, K., Van Merriënboer, B., Gulcehre, C., Bahdanau, D., Bougares, F., Schwenk, H., & Bengio, Y. (2014). Learning phrase representations using RNN encoder-decoder for statistical machine translation. arXiv preprint arXiv:1406.1078.

Dey, R., & Salem, F. M. (2017, August). Gate-variants of gated recurrent unit (GRU) neural networks. In 2017 IEEE 60th International Midwest Symposium on circuits and systems (MWSCAS) (pp. 1597-1600). IEEE.

Ding, Yanzhuo & Liu, Yang & Luan, Huanbo & Sun, Maosong. (2017). Visualizing and Understanding Neural Machine Translation. 1150-1159. 10.18653/v1/P17-1106.

Graves, A. (2014). Generating sequences with recurrent neural networks. arXiv preprint arXiv:1308.0850.

Jeblad. (2018a, February 8). File:Gated Recurrent Unit, type 1.svg - Wikimedia Commons. **https://commons.wikimedia.org/wiki/File:Gated_Recurrent_Unit,_type_1.svg**

Jeblad. (2018b, February 8). File:Gated Recurrent Unit, type 2.svg - Wikimedia Commons. https://commons.wikimedia.org/wiki/File:Gated_Recurrent_Unit,_type_2.svg

Jeblad. (2018c, February 8). File:Gated Recurrent Unit, type 3.svg - Wikimedia Commons. **https://commons.wikimedia.org/wiki/File:Gated_Recurrent_Unit,_type_3.svg**

Kulshrestha, N. (2020). Use of Deep Learning methods such as LSTM and GRU in polyphonic music generation. **https://norma.ncirl.ie/4451/1/nipunkulshrestha.pdf**

Mikolov, T. (2012). Statistical language models based on neural networks. Brno University of Technology, Faculty of information technology, Department of computer graphics and multimedia. **https://www.fit.vutbr.cz/~imikolov/rnnlm/thesis.pdf**

Pace, R. K., & Barry, R. (1997). Sparse spatial autoregressions. Statistics & Probability Letters, 33(3), 291-297.