Azure-ML Code Benutzungsanleitung

Übersicht der Projektstruktur

Auflistung der Verzeichnisse:

- /CMaps: Trainings- und Testdaten
- /documents: versch. Dokumente: dieses, PoC-Bericht, Paper über Datensatz
- /results: Messergebnisse des Experiments
- /graphs: Plots
    - /2d comparisons: Punktgrafen mit Achsen: val-acc / runtime
    - /epoch time distributions: Boxplots der Runtimes der einzelnen Architekturen
    - /lr comparisons: Lineplots zum Verleich versch. LRs je Modell
    - /old graphs: alte Plots des falsch durchgeführten Experiments
    - /proof of concept: Plots zum PoC-Experiment, inkl. Modellgrafiken
    - /run comparisons: Lineplots zum Vergleich der versch. Runs je Modell

    Wichtig sind insbesondere die Verzeichnisse:

    - 2d comparisons
    - lr comparisons
    - run comparisons

    Diese enthalten die aktuell wichtigen Plots und sind jeweils unterteilt in folgende Unterordner:

    - old: alte Versionen der Plots (mit anderer Auswahl an Modellen)
    - sliced-model: Plots für Sliced-Modelle
    - reference-model: Plots für (custom) Referenz-LSTM Modelle

    In den Verzeichnissen „/lr comparisons“, bzw. „/run comparisons“ hat jedes Modell einen eigenen Unterordner, da je nach verwendeter Aggregatfunktion (mean / nmedian / k-best („peak val“ genannt)) verschiedene Plots entstehen.

Codedateien liegen aktuell einfach im Hauptverzeichnis des Projektes. Dies sollte in Zukunft in einen /Code Verzeichnis refactured werden. **Um die Codefunktionalität vor dem Vortrag nicht zu brechen, wird aktuell empfohlen jede Datei exakt an ihrem Platz zu lassen, da der Code eine Vielzahl hart-codierter relativer Verzeichnisangaben enthält!**

Übersicht der wichtigen Codedateien:

- plot.py: Skripte zur Erzeugung der Plots in /graphs
- torch_custom_lstm.py: Sliced-LSTM Architektur als Klasse SlicedLSTM
- keras_proof_of_concept.py: PoC Experiments mittels Keras
- custom_lstm_example.py: Referenzarchitektur als Klasse CustomLSTM
- torch_lstm_experiment.py: Experiment inkl. Datenvorverarbeitungspipeline
- torch_current_model.py: Aktuelles (Sliced-/Referenz-) Modell für das Experiment.

Starten eines eigenen Experimentruns

Übersicht

Für einen Run des Experiments müssen in angegebener Reihenfolge folgende Schritte ausgeführt werden:

1. Definition des aktuellen Modells in torch_curent_model.py
    a. Layerstruktur in der entsprechenden Klasse verändern um neue Modellstruktur zu erzeugen
    b. Forward-Methode an neue Layerstruktur anpassen
2. torch_lstm_experiment.py anpassen:
    a. (Zeile 352) model_type einstellen: „reference-model“ oder „sliced-model“
    b. (Zeile 353) Architecture_string einstellen: sollte Modelllayout widerspiegeln
3. Torch_lstm_experiment.py ausfüren:
    Variante 1: via Pytorch (Rechtsklick, ausführen oder Strg+Umschalt+F10)
    Variante 2: via cmd / powershell:
        Windows: python <hier Dateipfad einfügen>\torch_lstm_experiment.py

Im Folgenden werden die einzelnen Schritte genauer erklärt:

Definition des aktuellen Modells

Das aktuelle Modell ist in der Datei torch_current_model.py definiert. Hier befindet sich je eine Klasse, um ein aktuelles Sliced- bzw. Referenzmodell zu definieren. Ein Experimentrun arbeitet nur mit einem Modell, entsprechend muss nur eine Klasse angepasst werden.

Veränderung des Sliced-LSTM Modells

Das SlicedLSTM ist in der Klasse SlicedModel definiert. Dieses besteht aus Sliced-Layern und Dropout-Layern.

Sliced-Layer bearbeiten

Hier ein Beispiel zur Definition eines Sliced-Layers:

Zunächst Liste mit Slice-Konfigurationen definieren: `slices_1 = [(12, 4), (13, 4)]`
Neues LSTM-Layer erstellen und als Klassenvariable speichern:

```
self.sliceLSTM1 = custom.SlicedLSTM(slices_1)
```

Das „self“ Präfix bewirkt, dass die Variable als Objektvariable gespeichert wird. Hier ist „sliceLSTM1“ der Name des aktuellen Layers. Das Layer ist stets eine Instanz der „SlicedLSTM“-Klasse, die mittels „custom“-Namen in die Codedatei importiert ist. Neu hinzugefügte Variablen sollten eindeutige neue Namen erhalten (z.b. slices_3, self.sliceLSTM3, …).

Die Codezeile:

```
last_hidden_units = sum(item[1] for item in slices_1)
```

sollte nach jeder Definition eines Sliced-Layers gestellt werden. **Evtl. den Namen slices_1 an den tatsächlichen Namen des Slice-Liste anpassen.** Hier wird die Variable last_hidden_units aktualisiert. Diese bewirkt, dass am Output-Layer niemals Änderungen vorgenommen werden müssen. (Mehr dazu unter Output-Layer).

Aktuell besteht das SlicedLSTM aus einem Sliced-Layer und einem Dropout-Layer. Soll lediglich die Slice-Aufteilung geändert werden, so reicht es die Liste `slices_1` neu zu definieren.

Um ein zweites Slice-Layer hinzuzufügen ist bereits ein Codesegement auskommentiert, welches einkommentiert geschlossen werden kann.

Folgt ein Sliced-Layer (G) einem Sliced-Layer (F) so muss stets sichergestellt werden, dass die total_input_size, also die Summe aller Input_Sizes aller Slices in (G) gleich der total_hidden_size von (F) also der Summe aller Hidden_Sizes alles Slices von (F) ist.

Die Summe aller Input_sizes dess ersten Sliced_Layers muss stets 25 sein (= Feature_size)!

Änderungen der Modellstruktur, bei der neue Layer hinzugefügt oder Layer entfernt werden müssen ebenfalls in der forward-Methode das Klasse abgebildet werden!

Dropout-Layer bearbeiten

Das Dropoutlayer ist aktuell wie folgt definiert:

```
self.DropOut1 = nn.Dropout(p=0.20)
```

Wobei der Parameter p die Dropoutwahrscheinlichkeit bestimmt (p=0.20 entspricht 20%). Um die Dropoutwahrscheinlichkeit zu ändern reicht es aus, den Wert von p zu verändern.

Um das Dropoutlayer zu entfernen, die entsprechende Codezeile durch vorangestellte „#“ auskommentieren.

Änderungen der Modellstruktur, bei der neue Layer hinzugefügt oder Layer entfernt werden müssen ebenfalls in der forward-Methode das Klasse abgebildet werden!

Ein neues Dropoutlayer kann z.b. durch folgende Codezreile hinzugefügt werden:

```
self.DropOut2 = nn.Dropout(p=0.10)
```

Änderungen der Modellstruktur, bei der neue Layer hinzugefügt oder Layer entfernt werden müssen ebenfalls in der forward-Methode das Klasse abgebildet werden!

Anmerkung:

Ein Dropoutlayer sollte stets nach einem LSTM-Layer folgen. Zwei aufeinanderfolgende Dropoutlayer sind nicht sinnvoll, da sie perfomanter durch ein Dropoutlayer mit multiplizierter Dropoutwahrscheinlichkeit ausgedrückt werden können:

Dropoutlayer 1 p1 = 0.2 -> Dropoutlayer 2 mit p2=0.25 können modelliert werden durch ein Dropoutlayer mit p = (1-p1)*(1-p2) = (1-0.2)*(1-0.25) = 0.8*0.75 = 0.6

Anpassung der Forward-Methode

In der Forward-Methode der Modelklasse wird die genaue Bearbeitung der Inputdaten festgelegt. Änderungen der Forward-Methode sind immer dann notwendig, wenn die Modellstruktur so geändert wurde, dass Layer hinzugefügt oder entfernt wurden. Sonst werden Änderungen ggf. nicht berücksichtigt oder es führt zu einem Absturz des Programms.

Beispielhafte Fotrward-Methode:

```
def forward(self, x):
    x, _ = self.sliceLSTM1(x)
    x = x[:, -1, :]  # only take last hidden state (equals "return_sequence
= False")
    x = self.DropOut1(x)

    # x, _ = self.sliceLSTM2(x)
    # x = x[:, -1, :]  # only take last hidden state (equals
"return_sequence = False")
    # x = self.DropOut2(x)

    x = self.out(x)
    x = self.out_activation(x)
    return x
```

Der Parameter x bezeichnet die Inputdaten.

Die Inputdaten werden in der Dimension [Batchsize, Sequence_Length, Feature_Length] übergeben.

Damit ein Layer der Modellklasse tatsächlich verwendet wird, muss es in der Forward-Methode aufgerufen werden. Sonst wird es bei der tatsächlichen Daten-berechnung außen vor gelassen.

Im Beispiel bewirkt:

```
x, _ = self.sliceLSTM1(x)
```

dass die Daten das erste Sliced-Layer durchlaufen. Ein Sliced-Layer hat immer zwei Rückgabewerte, von denen nur der erste berücksichtigt werden muss, daher das Abspeichern via: `x, _ =`

Ein Sliced-Layer erzeugt als Output eine Sequenz (von Hidden-states der Zelle). Dieser Output kann so ohne Weiteres an ein nachfolgendes Sliced-Layer übergeben werden. Folgt jedoch in der nach einem Sliced-Layer kein weiteres Sliced-Layer in der Berechnung, so sollte diee Sequenz abgeschnitten werden, sodass nur der letzte Hidden-State als Output übergeben wird.

Das passiert mit der Zeile:

```
x = x[:, -1, :]  # only take last hidden state (equals "return_sequence = False")
```

(Es ist egal ob ein nachfolgendes Sliced-Layer unmittelbar nach dem ersten kommt oder erst ein Dropout-Layer dazwischen liegt. Folgt irgendwann ein Sliced-Layer so sollte die Sequenz nicht abgeschnitten werden.)

Diese Codezeile sollte also **nach dem letzten Sliced-Layer Aufruf** insgesamt genau einmal in der Forward-Methode sein.

Diese auskommentierten Zeilen zeigen, wie die Forward-Methode angepasst werden müsste, falls ein zweites Sliced-Layer hinzugefügt wird:

```
# x, _ = self.sliceLSTM2(x)
# x = x[:, -1, :]  # only take last hidden state (equals "return_sequence =
```

```
False")
# x = self.DropOut2(x)
```

In diesem Fall muss jedoch die Zeile:

```
x = x[:, -1, :]  # only take last hidden state (equals "return_sequence = False")
```

weiter oben im Code auskommentiert / entfernt werden, sodass sie insgesamt genau einmal und zwar hinter dem letzten Sliced-Layer steht.

Das Output-Layer

Die Modellklasse beinhaltet auch ein Output-Layer, welches am Ende der Forward-Methode aufgerufen wird. **Dieses Output-Layer sollte nicht verändert werden.** Für eine korrekte Berechnung des Modells ist es zwingend notwendig, dass das Output-Layer bei der Definition die Größe gesamte Hidden-Size des letzten vorangegangenen Sliced-Layers bei der Definition als Parameter übergeben bekommt.

Sofern die Codezeile:

```
last_hidden_units = sum(item[1] for item in <Hier-Name-der-List-of-Last-Slice-Layer-einfügen>)
```

nach jeder Definition eines Sliced-Layers gesetzt wurde muss hierzu nie eine Anpassung vorgenommen werden.

Änderung der Referenz-Klasse

Eine Dokumentation zu Änderungen der Referenz-Klasse wird bei Bedarf nachgereicht.