Programmdokumentation

Room2IsoSpace

Abschlussaufgabe Gruppe 4

Kristina Matz

Leonidas Devetzidis

Chantal Klemm

18.03.2021

Aufgabe 1 – QSLinks

1.1. Aufgabenstellung

Anhand der erzeugten Matterport3d Daten soll ein Feed-Forward Netzwerk trainiert werden, welches anhand von Objekttypen die QSLinks zwischen diesen bestimmt.

1.2. Umsetzung/Trainingsdaten

Von Tim Hau wurde aus dem Matterport 3D Datensatz eine xml-Datei erstellt, in der die Spatial Entities und QSLinks einer House Datei des Datensatzes dargestellt werden.

Aus der xml-Datei ’Matterport3dhouse.xml’ werden die Trainingsdaten für das Netz generiert. Hierzu wird jeder QSLink der Datei dargestellt als [figure, ground, relType]. ’figure’ und ’ground’ bekommen für das Training die ’objectId’ der zugehörigen Spatial-Entity zugewiesen. Da für uns nur der Typ des Objekts, also das zugehörige Label interessant ist, bekommen Objekte mit verschiedenen ID’s aber dem gleichen Label die gleiche ID zugewiesen.

QSLinks des Typ NTTPc/TPPc werden zum Typ NTPP/TPP, indem ‘figure’ und ‘ground’ getauscht werden.

In den vorliegenden Trainingsdaten kommen nur die drei QS-Relationen ’PO’, ’NTPP’ und ’EC’ vor. Diese bekommen in der genannten Reihenfolge die ID’s 0, 1 und 2.

figure und ground sind die beiden Objekte und somit die Eingabe, die das Netz erhält, relType ist der zugehörige QSLink und somit die Ausgabe (bzw. die jeweilige ID).

Für ein besseres Training werden die Trainingsdaten balanciert, sodass von jedem Link-Typ 999 Beispiele (wenn möglich) zufällig genutzt werden. Es liegen nur 37 ‘EC’ Links vor, deshalb müssen diese vervielfacht werden, damit auch von ihnen 999 vorliegen. Diese Trainingsdaten werden dann zufällig in ein Trainings- und Validierungsset aufgeteilt.

Anschließend werden die Trainingsdaten unterteilt in X_train und y_train, wobei X_train den Input für das Netz enthält und y_train das korrekte Ergebnis des Inputs. Die Validierungsdaten werden nach demselben Prinzip unterteilt in X_val und y_val.

Damit unser Netz diese Daten als Eingabe nutzen kann, müssen sie zu Tensoren umgewandelt werden. Dazu werden aus den Listen erst Arrays erstellt, wobei die Inputs dabei auch normalisiert werden.

Die beiden Arrays der Validierungsdaten werden direkt zu Tensoren umgewandelt.

Für die Trainingsdaten nutzen wir die definierte Klasse Data, mit deren Hilfe wir einen trainloader erstellen und so während dem Training leicht durch die Daten iterieren können. In der __init__(self) Methode der Klasse werden dann die beiden Arrays der Trainingsdaten zu Tensoren umgewandelt.

Allgemein ist darauf zu achten, dass die Werte der Input Daten in den Arrays den Datentyp float haben und die Tensoren der Ergebnisse den Datentyp long haben.

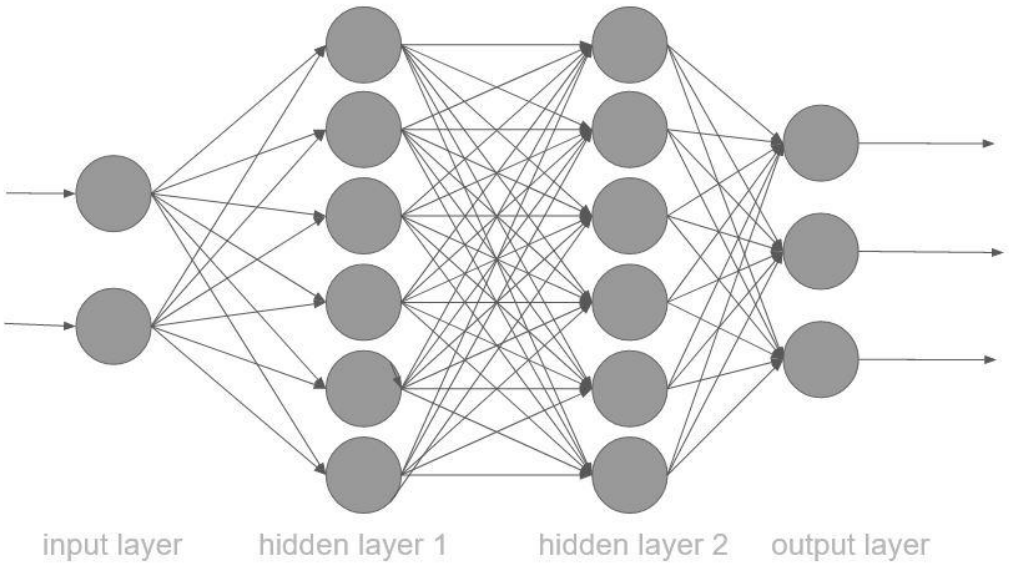
1.3.1. Implementierung

Als Programmiersprache wird Python genutzt. Um das neuronale Netz aufzusetzen wird die PyTorch Bibliothek genutzt.

Verwendete Bibliotheken:

- PyTorch
- NumPy
- ElementTree
- sklearn
- matplotlib
- random
- sys
- os

Neuronales Netz



Das Netz soll eine von drei möglichen Klassen (QSLinks) für die Eingabe [*figure, ground]* vorhersagen. Aus diesem Grund haben wir uns dafür entschieden ein Neuronales Netz für multiclass classification zu nutzen.

Unser Netz hat insgesamt vier Layer, wobei wir zwei Input Neuronen haben, da wir zwei Objekte als Eingabe bekommen. Für multiclass classifier ist die Anzahl der Output Neuronen gleich der Anzahl der möglichen Klassen. Da es in unseren Trainingsdaten drei mögliche QSLinks gibt, sind dies also drei Output Neuronen.

So wie es im Allgemeinen üblich ist für multiclass classifier, werden die Layer mit ihren weights und biases in der __init__() Methode definiert.

Die __init__() Methode beginnt damit, dass drei Schichten von Neuronen initialisiert werden.

```
def __init__(self):
    super(NeuralNet, self).__init__()
    # Schichten
    self.lin1 = nn.Linear(2, 6) #2-(6-6)-3
    self.lin2 = nn.Linear(6, 6)
    #output layer: [Klasse PO, Klasse NTPP, Klasse EC]
    self.oupt = nn.Linear(6, 3)
```

Wichtig ist hierbei zu erwähnen, dass nicht explizit ein Input Layer definiert wird, da keine Bearbeitung auf den Eingabewerten stattfindet.

Die Linear() Klasse definiert eine komplett vernetzte Netzwerkschicht.

Die weights und biases des Netzwerks werden folgendermaßen definiert:

```
    nn.init.xavier_uniform_(self.lin1.weight)
    nn.init.zeros_(self.lin1.bias)
    nn.init.xavier_uniform_(self.lin2.weight)
    nn.init.zeros_(self.lin2.bias)
    nn.init.xavier_uniform_(self.oupt.weight)
    nn.init.zeros_(self.oupt.bias)
```

Jede Schicht hat weights, welche sie zu der vorherigen Schicht vernetzt. Man kann also sagen, dass self.lin1.weight eine Matrix von den weights der Input Neuronen zu den Neuronen in der lin1 Schicht, self.lin2.weight eine Matrix der weights von den lin1 Neuronen zu den lin2 Neuronen, und self.oupt.weight eine Matrix der weights der lin2 Neuronen zu den Output Neuronen ist.

Die weights und biases werden durch PyTorch automatisch initialisiert. Unser Programm nutzt xavier_uniform_() als Initialisierung aller weights und es initialisiert alle biases als 0.

Da das Netz als Klasse definiert wurde, muss eine forward() Methode definiert sein, welche Input Tensoren bekommt und Output Tensoren berechnet.

```
def forward(self, x):
    z = torch.tanh(self.lin1(x))
    z = torch.tanh(self.lin2(z))
    z = self.oupt(z)  # no softmax: CrossEntropyLoss()

    return z
```

Der Parameter x ist ein batch mit einem oder mehreren Tensoren. Die lin1 Schicht bekommt nacheinander x als Input und die Aktivierungsfunktion tanh() wird angewendet. Das Ergebnis wird als ein neuer Tensor z zurückgegeben. Tanh() sorgt dafür, dass alle Neuronen von lin1 Werte zwischen -1.0 und +1.0 annehmen. Der neue Tensor z wird dann als Eingabe an lin2 übergegeben und es wird ebenfalls Tanh() angewendet. Der daraus erzeugte neue Tensor z wird der Output Schicht als Eingabe gegeben, hier jedoch ohne Aktivierungsfunktion. Wir nutzen an dieser Stelle keine Aktivierungsfunktion, da wir bei dem Trainieren des Netzes die CrossEntropyLoss() Funktion nutzen. Diese Funktion wendet automatisch die Softmax() Aktivierungsfunktion in Form einer speziellen LogSoftmax() Funktion an.

Das fertig definierte Netz als Klasse ohne Parameter kann dann durch net = NeuralNet() instanziiert werden.

Training

Als optimizer nutzen wir SGD mit einer initialisierten learning rate von 0.01.

Das Netz wird in 100 Epochen trainiert.

In jeder Epoche werden einmal die Trainingsdaten durchlaufen.

Für jedes Element i in den Trainingsdaten wird jeweils einmal die zero_grad() Funktion aufgerufen, welche die gradients aller optimerten Tensoren auf 0 zurücksetzt.

Anschließend wird i dem Netz als input übergeben und so eine Vorhersage berechnet.

Mit Hilfe dieser berechneten Vorhersage des Ergebnisses und dem echten Ergebnis, wird der Loss berechnet. Als Funktion zur Berechnung des Loss benutzen wir, wie

bereits erwähnt, CrossEntropyLoss(). Dadurch wird später die Wahrscheinlichkeit pro Klasse vorhergesagt.

Anschließend werden durch die .backward() Funktion die gradients berechnet und durch die .step() Funktion die Parameter damit geupdatet.

Der Aufbau der Training-Schleife:

```
# epochs
for epoch in range(100):
    for x, y in trainloader:
        # clear gradient
        optimizer.zero_grad()
        # make a prediction
        z = model(x)
        # calculate loss, da Cross Entropy benutzt wird Wahrscheinlichkeit pro Klasse
        # vorhergesagt. Das macht torch.max(y,1)[1])
        loss = criterion(z, y)
        # calculate gradients of parameters
        loss.backward()
        # update parameters
        optimizer.step()
```

Während dem Training speichern wir den Loss und berechnen sowohl die trainings accuracy als auch die validation accuracy. Trainings accuracy und validation accuracy werden ebenfalls gespeichert.

Accuracy

Zur Bestimmung der accuracy haben wir eine Funktion accuracy(y_hat, y) definiert. Der Parameter y_hat ist dabei das vorhergesagte Ergebnis des Netzes und der Parameter y das korrekte Ergebnis.

```
def accuracy(y_hat, y):
    pred = torch.argmax(y_hat, dim=1)
    return (pred == y).float().mean()
```

Die Funktion argmax() gibt den Index des größten Wertes zurück. Damit wird also die Klasse bestimmt, die das Netz vorhergesagt hat. Diese wird mit dem korrekten Ergebnis verglichen.

Anschließend lassen wir noch einen “classification report” berechnen. Dieser zeigt die wichtigsten Klassifizierungs-Metriken an. Es wird also precision, recall und der F1 Score berechnet.

```
              precision    recall  f1-score   support

          PO       0.41      0.75      0.53       589
        NTPP       0.94      0.48      0.64       593
          EC       0.45      0.30      0.36       616

    accuracy                           0.51      1798
   macro avg       0.60      0.51      0.51      1798
weighted avg       0.60      0.51      0.51      1798
```

Die Funktion classification_report() bekommt als Eingabe die korrekten Ergebnisse, die berechneten Ergebnisse des Netzes und eine Liste mit den Namen der Klassen (PO, NTPP, EC).

1.3.2. Anwenderdokumentation

Das zugehörige Programm ist Aufgabe1.py im github. Um das Programm auszuführen sollte die matterport3dhouse.xml im gleichen Ordner wie Aufgabe1.py liegen. Außerdem sollte die requirements.txt installiert werden.

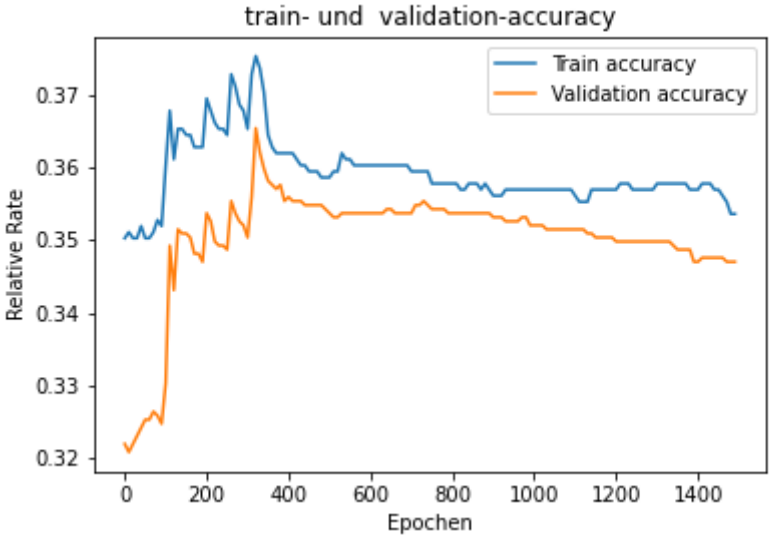
```
# using pip
pip install -r requirements.txt

# using Conda
conda create --name <env_name> --file requirements.txt
conda install --file requirements.txt
```

Beim Ausführen des Programms wird das Netz trainiert, gespeichert und der Trainingsverlauf graphisch dargestellt.

1.4. Hyperparameteranalyse und Trainingsverlauf

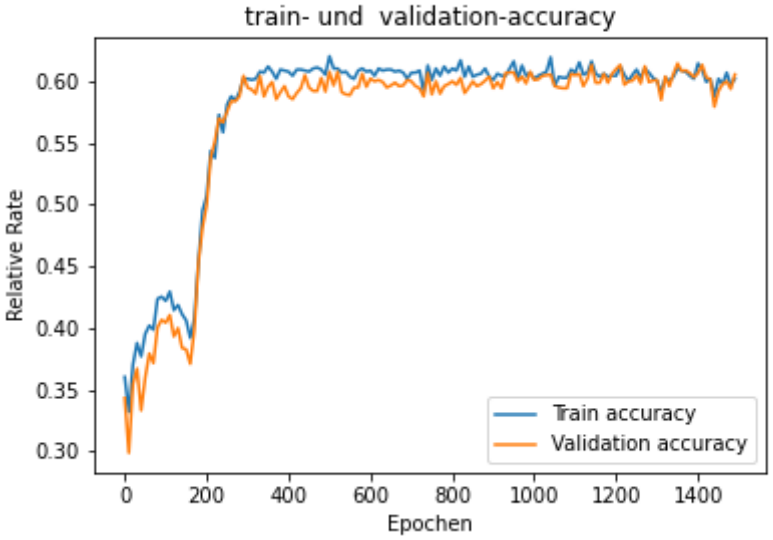
Im Folgenden werden Ergebnisse des implementierten Modells vorgestellt. Anhand der Ergebnisse soll eine effiziente Kalibrierung der Hyperparameter des Modells an die Aufgabenstellung erfolgen. Dabei wird der Fokus auf die trainings- bzw. validation-accuracy und den f1-score gelegt. Die angegebenen Laufzeiten beziehen sich dabei auf einen AMD Ryzen 7 3700X Prozessor mit 32GB DDR4-3600 Arbeitsspeicher. Das eingesetzte Betriebssystem ist Windows 10 64bit Build 18363.1440.

Der nachfolgende Graph zeigt die trainings- und validation-accuracy über 1500 Epochen des zuerst vorgestellten Modells.



Pro Epoche wurden einmal alle Daten gelernt bzw. durch den Forward- und Back-Propagation Prozess geschickt. Bei steigender Epochenanzahl steigt die train- und validation-accuracy unruhig bis kurz vor der 400. Epoche auf ca. 37% an. Danach sinken sie auf ungefähr 35% ab. Beide Kurven verhalten sich hier ziemlich gleich. Die Kurven zeigen ein minimales Overfitting, da die train-accuracy leicht über der validation-accuracy steht.

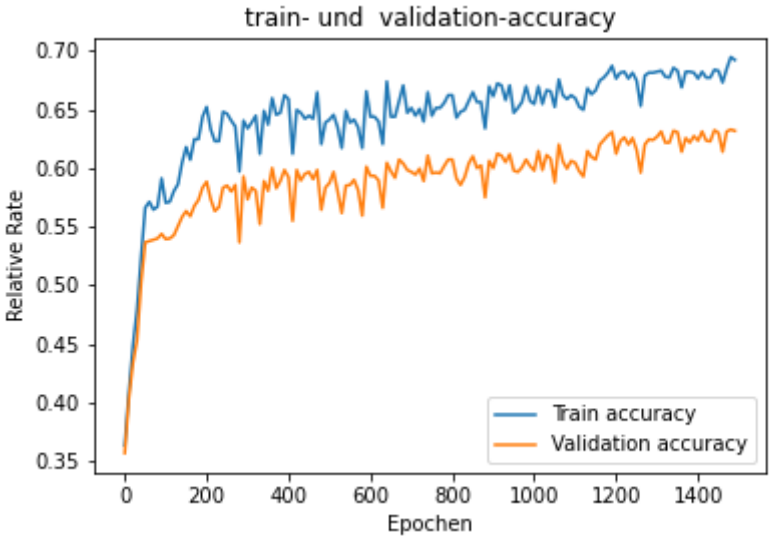
Im ersten Schritt wird SGD durch adam mit einer initialen learning rate von 0,01 als optimizer ausgetauscht, da sich der Adamoptimizer im Bereich des Deep Learnings hinsichtlich der Performance etabliert und damit SGD abgelöst hat. Der nachfolgende Graph zeigt den Performance-Zuwachs durch diesen an.



Der Lernprozess verläuft hier deutlich ruhiger. Beide Kurven steigen gleichmäßig bis zur 100. Epoche auf ca. 42% an. Anschließend fallen beide auf ca. 37% kurz vor der 200. Epoche. Ab der 200. Epoche steigen beide accuracies gleichmäßig exponential

an. Kurz vor der 400. Epoche pendeln sich beide accuracies auf ca. 60% bis zur letzten Epoche ein. Die Aktivierungsfunktionen werden von Tanh auf ReLU geändert, da ReLU in den meisten neuronalen Netzwerken die empfohlene Aktivierungsfunktion ist. Der Accuracygraph zeigt jedoch keine nennenswerte Veränderung.

Als nächstes wird die Anzahl der Neuronen in den Hiddenlayer solange verdoppelt, bis sich kein Performance-Zuwachs mehr bemerkbar macht.



Schon bei der ersten Verdoppelung von 6 auf 12 Neuronen ist Overfitting im Graph sichtbar. Um dem entgegenzuwirken wird ein Dropoutlayer mit p = 0.5 zwischen dem Outputlayer und dem Hiddenlayer platziert. Dadurch zeigen beide Kurven wieder ungefähr die gleiche accuracy. In der anschließenden Tabelle zeigt sich der Performance-Zuwachs mit Verdoppelung der Neuronenanzahl und zusätzlichen Hiddenlayern, anhand eines durchschnittlichen f1-scores über alle drei Klassen: macro avg(f1).

#Hiddenlayer	#Neuronen	macro avg(f1)	Laufzeit(min.)
2	6	0.62	0.63
2	12	0.62	0.64
2	24	0.67	0.64
2	48	0.72	0.65

2	96	0.74	0.70
2	192	0.71	0.84
2	384	0.66	1.32
2	768	0.69	2.75
3	96	0.62	1.64
3	384	0.63	4.02
3	768	0.65	8.13
3	1536	0.62	19.92
4	192	0.32	2.50

Es wird deutlich, dass das Modell mit zwei Hiddenlayer und 96 Neuronen mit einer macro avg(f1) von 0.74 am besten performt. Ab diesem Punkt schadet das sukzessive Vergrößern der Architektur eher der Performance des Modells. Eine mögliche Erklärung für diese Beobachtung könnte die Schwierigkeit des Klassifizierungsproblems sein. Wenn diese sehr gering ist und eine größere Architektur verwendet wird als nötig, könnte sich dies kontraproduktiv auswirken. Mehr Layer würden eine viel zu komplexe hochgradige polynomiale Separierung im Punkteraum vornehmen. Ist dann der Punkteraum, jedoch z.B. linear separierbar würde die komplexe Funktion versagen.

1.5. Evaluationsergebnisse

	precision	recall	f1-score	support
PO	0.80	0.47	0.59	592
NTPP	0.76	0.81	0.79	606
EC	0.73	0.98	0.83	600
accuracy			0.75	1798
macro avg	0.76	0.75	0.74	1798
weighted avg	0.76	0.75	0.74	1798

Zu sehen ist der classification report des am besten performenden Modells mit 96 Neuronen, der mithilfe der scikit-learn Bibliothek erstellt wurde. Es fällt auf, das der f1-score für PO Relationen mit 0.59 am schlechtesten performt. Dies könnte unter anderem auf Dysbalancen in den Trainingsdaten zurückzuführen sein. NTPP und EC Relationen performen mit 0.79 und 0.83 vergleichsweise gut. Bei den EC Relationen könnte dies aufgrund der geringen vielfalt in den EC Daten herrühren. Der f1-score errechnet sich aus dem Harmonischen Mittel zwischen precision und recall.

1.6. Verbesserungsmöglichkeiten

In der xml Datei aus der wir die Trainingsdaten generiert haben gibt es insgesamt 6456 QSLinks, von diesen sind jedoch nur 37 vom Typ ‘EC‘. Eine größere Anzahl an Beispielen für diese könnte das Netz verbessern. Desweiteren performt unser neuronales Netzwerk mit bis zu 20% performanceunterschied, wenn es mit unveränderter Architektur mehrmals gestartet wird. Eine mögliche erklärung wären Dysbalancen in den Trainingsdaten, da stets eine zufällige Teilmenge der Daten bei jedem start geladen werden. Eine ausführlichere Hyperparameteranalyse könnte die performance des Modells verbessern. Eine weitere Möglichkeit, wäre das Erstellen einer ROC-Kurve. Wichtig wäre auch die Generalisierungsfähigkeit der vorgestellten Modelle auf einen Testdatensatz, der aus einer anderen und unabhängigen Quelle stammt. Andernfalls könnte ein Overfitting der Modelle nicht ausgeschlossen werden.

Aufgabe 2 – Objekt + QSLinks

2.1. Aufgabenstellung

Es sollte ein weiteres Netzwerk trainiert werden, welches anhand eines Eingabeobjekts weitere Objekte inklusive der zugehörigen QS-Relation vorschlägt.

2.2. Umsetzung/Trainingsdaten

Grundlage für die Trainingsdaten sind wieder die Daten aus der matterport3dhouse.xml. Das nun zu trainierende Netz soll jedoch als Eingabe nur ein Objekt erhalten und dann alle möglichen weiteren Objekte und den zugehörigen QSLink ausgeben.

Hierfür wird [figure, ground, relType] aufgeteilt in ‘figure‘ als Eingabe und ‘ground‘,‘type‘ als eine mögliche Ausgabe. Die gesamte Ausgabe für ‚figure‘ sollen dann alle möglichen ‘ground‘,‘type‘ Paare sein.

So entstehen 78 mögliche Ausgaben. Diese Ausgaben werden von 0 bis 77 durchnummeriert. Jedes Objekt (‘figure’) erhält nun eine Liste der Länge 78. An der Stelle i der Liste eines Objekts steht eine 1, wenn das zugehörige Paar ’ground’, ’relType’ eine mögliche Ausgabe für das Objekt ist. Ist es keine mögliche Ausgabe steht an Stelle i eine 0.

Die so generierten Trainingsdaten werden dann in Trainings- und Validierungsdaten aufgeteilt. Anschließend werden die Trainingsdaten unterteilt in X_train und y_train, wobei X_train den Input für das Netz enthält und y_train das korrekte Ergebnis des Inputs enthält. Die Validierungsdaten werden nach demselben Prinzip unterteilt in X_val und y_val.

Damit unser Netz diese Daten als Eingabe nutzen kann, müssen sie zu Tensoren umgewandelt werden. Dazu werden aus den Listen erst Arrays erstellt und diese dann zu Tensoren umgewandelt. Hierbei ist noch zu beachten, dass der benötigte Datentyp in den Arrays der Inputs float ist und die Tensoren von y_train und y_val den Datentyp long haben.

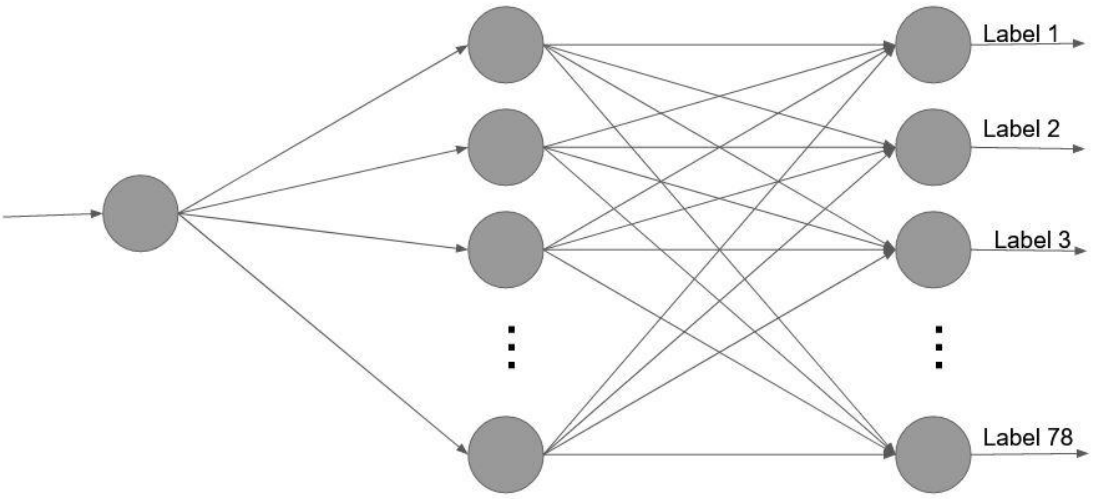
2.3.1. Implementierung

Auch für diese Aufgabe wird Python und PyTorch genutzt.

Verwendete Bibliotheken:

- PyTorch
- NumPy
- ElementTree
- sklearn
- matplotlib
- random
- sys
- os

Neuronales Netz



Da für ein Objekt jeweils mehrere Objekte vorhergesagt werden können, haben wir uns in dieser Aufgabe dafür entschieden ein Neuronales Netz für multilabel classification zu verwenden. Hierbei bekommt jedes Objekt, welches eine mögliche Eingabe sein kann, einen festen Vektor zugeteilt, der die Länge der verschiedenen möglichen Ausgaben, also Labels, hat. Der Vektor enthält eine 1 an jeder Stelle, die für die Eingabe zutrifft. An allen anderen Stellen steht eine 0.

Unser Netz hat insgesamt drei Layer, wobei wir ein Input Neuron haben, da wir immer ein Objekte als Eingabe bekommen. Für multi label classifier ist die Anzahl der Output Neuronen gleich der Länge des Ausgabe Vektors. Aus unseren Trainingsdaten ergibt sich für die Anzahl der möglichen Ausgaben 78. Dadurch haben die Ausgabe Vektoren der verschiedenen Objekte die Länge 78 und somit ist unsere Anzahl der Output Neuronen 78.

Die __init__() Methode beginnt damit, dass drei Schichten von Neuronen initialisiert werden.

```
def __init__(self, nlabel):
    super(Net, self).__init__()
    self.main = nn.Sequential(
        nn.Linear(1, 64),
        nn.ReLU(),
        nn.Linear(64, nlabel),
    )
```

Der Parameter nlabel entspricht der Länge des Ausgabe-Vektors.

Anders als in Aufgabe 1 benutzen wir hier allerdings die Sequential Klasse. Dadurch wird das Netz gleichzeitig definiert und erstellt. Durch diesen Ansatz werden weight und bias nicht explizit initialisiert, sondern es wird die default Initialisierung von PyTorch genutzt.

Die verwendete Aktivierungsfunktion ist ReLU().

Die Funktion ReLU gibt 0 zurück, falls sie eine negative Eingabe bekommt und für jede positive Eingabe gibt sie die Eingabe wieder zurück.

Durch Sequential muss man in der Regel keine forward() Methode definieren, da sie automatisch erstellt wird.

In unserem Programm haben wir allerdings eine forward() Methode definert, welche jedoch nur den input an self.main übergibt.

```
def forward(self, input):
    return self.main(input)
```

Das fertig definierte Netz als Klasse, mit dem Parameter nlabel definiert, kann dann durch classifier = Net(nlabel) instanziiert werden.

Training

Als optimizer nutzen wir Adam mit der default learning rate 0.001.

Das Netz wird in 100 Epochen trainiert.

In jeder Epoche werden einmal die Trainingsdaten durchlaufen.

Für i in den Trainingsdaten wird jeweils einmal die zero_grad() Funktion aufgerufen, welche die gradients aller optimerten Tensoren auf 0 zurücksetzt.

Anschließend wird i dem Netz als input übergeben und so eine Vorhersage berechnet.

Mit Hilfe dieser berechneten Vorhersage des Ergebnisses und dem echten Ergebnis, wird der Loss berechnet. Als Funktion zur Berechnung des Loss benutzen wir MultiLabelSoftMarginLoss(). Diese wird benötigt, da wir ein Netz für multi label classification nutzen. Deshalb ist es auch wichtig die Trainingsdaten, nachdem sie in Tensore umgewandelt wurden, umzuformen, um Fehler zu vermeiden. Damit geht man noch einmal sicher, dass die übergebenen Tensoren die richtige Größe haben.

Anschließend werden durch die .backward() Funktion die gradients berechnet und durch die .step() Funktion die Parameter damit geupdatet.

Der Aufbau der Trainings-Schleife:

```
epochs = 100
for epoch in range(epochs):
    losses = []
    hammingloss = []
    for i, sample in enumerate(X_train):
        inputv = torch.from_numpy(sample)    # array to tensor
        inputv = Variable(inputv).view(1, -1)    # for MultiLabelSoftMarginLoss()

        labelsv = torch.from_numpy(y_train[i]).long()    # array to tensor
        labelsv = Variable(labelsv).view(1, -1)    # for MultiLabelSoftMarginLoss()


        optimizer.zero_grad()

        output = classifier(inputv)
        loss = criterion(output, labelsv)

        loss.backward()
        optimizer.step()
```

Accuracy

Zur Bestimmung der accuracy muss als erstes wieder ein Vektor erstellt werden der nur Nullen und Einsen enthält.

Dafür wird auf den output des Netzes die Sigmoid Funktion angewendet. Dadurch wird der Wertebereich auf 0 bis +1 begrenzt. Um nun von diesem Vektor einen Vektor zu bekommen, welcher nur Nullen und Einsen enthält, benötigt es einem threshold. Diesen haben wir als 0.5 gewählt. Das heißt, dass alle Einträge des Vektors die größer als 0.5 sind zu 1 werden und alle Einträge die kleiner als 0.5 sind zu 0 werden.

Bei der multi label classification ist die falsche Vorhersage eines Labels nicht mehr eine komplett falsche Vorhersage, so wie es in Aufgabe 1 bei multi class classification der Fall war. Es können trotzdem andere Label richtig vorhergesagt worden sein.

Daher muss man bei der Bestimmung der Accuracy anders vorgehen als in Aufgabe 1.

Wir bestimmen den Hamming Loss, den Hamming Score und einen F1 Score.

Der Hamming Loss ist der Bruchteil der Labels die falsch vorhergesagt wurden. Je kleiner also der Hamming Loss wird, desto besser performt unser Netz. Ein Netz welches alle Label richtig vorhersagen würde hätte dementsprechend einen Hamming Loss von 0.

Der Hamming Score gibt an wie viele der Labels (in unserem Fall 78) richtig vorhergesagt wurden. Je höher also der Hamming Score ist, desto besser performt das Netz.

Der F1 Score kann als gewichteter Durchschnitt von precision und recall interpretiert werden. Je größer der F1 Score ist, desto besser performt das Netz. Ein perfekter F1 Score wäre 1. Im Allgemeinen wird der F1 Score bei multi label classification nicht benutzt. Da wir unseren Ausgabe Vektor jedoch binär kodiert haben, war es möglich ihn dennoch zu bestimmen. Als average nutzen wir ‘micro’, welcher die True Positives, False Negatives und False Positives zählt.

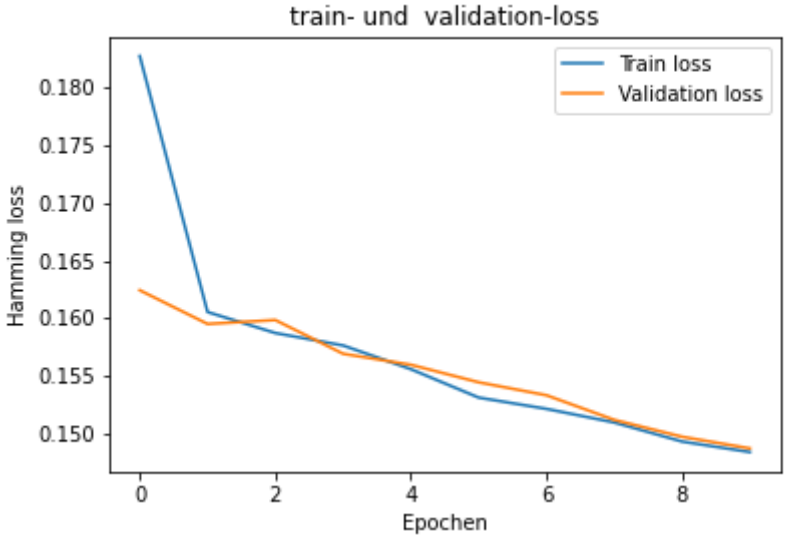
Alle drei Funktionen bekommen als Eingabe das korrekte Ergebnis und das vorhergesagt Ergebnis durch das Netz. Das vorhergesagte Ergebnis muss dafür als binär kodierter Vektor vorliegen, welches durch das oben beschriebene Schema passiert.

2.3.2 Anwenderdokumentation

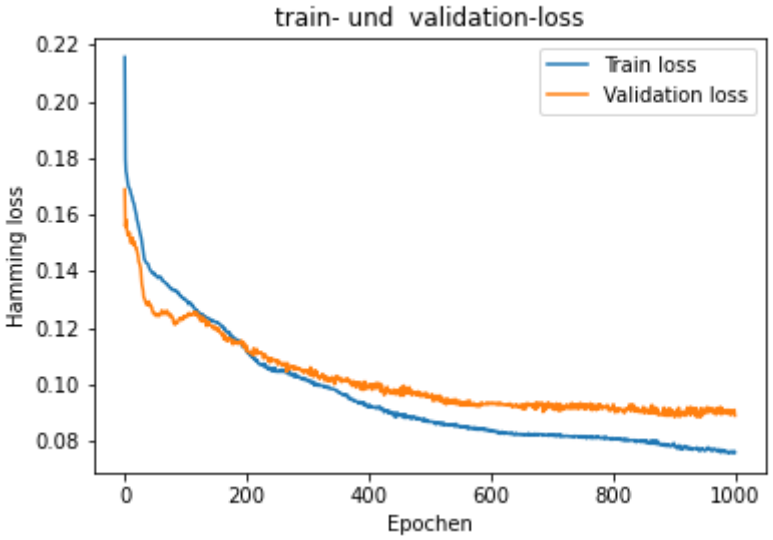
Das zugehörige Programm ist Aufgabe2.py im github. Um das Programm auszuführen sollte außerdem die matterport3dhouse.xml im gleichen Ordner wie Aufgabe2.py liegen. Außerdem sollte die requirements.txt installiert werden, falls dies noch nicht aus vorherigen Teilen geschehen ist.

Beim Ausführen des Programms wird das Netz trainiert, gespeichert und der Trainingsverlauf graphisch dargestellt.

2.4. Hyperparameteranalyse und Trainingsverlauf



Zu sehen ist der Hamming train- und validation-loss in jeder der Epochen. Beide sinken kontinuierlich bei ansteigender Epochen Anzahl bis zur letzten Epoche auf ca. 0.15. Zu Beginn ist der train-loss noch höher. Kurz vor der zweiten Epoche nehmen beide Kurven ähnliche Werte an und verhalten sich nahezu gleich. Aufgrund der Tatsache, dass beide Kurven bis zur letzten Epoche kontinuierlich sinken, wird im nächsten Schritt die Epochen Anzahl erhöht.



Durch die Erhöhung der Epochen Anzahl auf 1000, wird ein besserer Hamming Loss erreicht. Zu Beginn zeigt sich leichtes Underfitting. Ab der 200. Epoche ist das Gegenteil der Fall. Das Modell Overfittet bei steigender Epochen Anzahl immer

stärker. Folglich wird im nächsten Schritt dem Netzwerk ein Dropout mit p = 0.5 hinzugefügt, sodass sich die Werte der beiden Kurven aneinander anpassen. In der anschließenden Tabelle zeigt sich der Performance-Zuwachs mit Verdoppelung der Neuronenanzahl und zusätzlichen Hiddenlayern, anhand des Hamming Losses in der Validierungsphase.

#Hiddenlayer	#Neuronen	Hamming Loss	Laufzeit(min.)
1	64	0.138	7.4
1	128	0.133	7.6
1	256	0.122	8.0
1	512	0.108	9.1
1	1024	0.102	10.5
1	2048	0.102	13.4
1	4096	0.094	18.5
1	8192	0.092	27.5
1	16384	0.092	42.8
2	64	0.126	7.9
2	128	0.122	8.8
2	512	0.107	17.4
2	2048	0.149	120.2
2	8192	X	24h>

Das Modell mit 8192 Neuronen performt mit 0.092 und einer Laufzeit von 27.5min am effizientesten. Interessant wäre hier das weitere Erhöhen der Neuronen und Hiddenlayer. Leider gestaltet sich dies aufgrund von Zeit und Hardwarebeschränkungen als schwierig.

2.5. Evaluationsergebnisse

```
accuracy per Object:
Correct: 78 Percentage: 100.0 ObjectID: 0 Object: cabinet
Correct: 78 Percentage: 100.0 ObjectID: 1 Object: objects
Correct: 78 Percentage: 100.0 ObjectID: 2 Object: towel
Correct: 78 Percentage: 100.0 ObjectID: 3 Object: shower
Correct: 78 Percentage: 100.0 ObjectID: 4 Object: curtain
Correct: 78 Percentage: 100.0 ObjectID: 5 Object: bathtub
Correct: 78 Percentage: 100.0 ObjectID: 6 Object: wall
Correct: 77 Percentage: 98.7179 ObjectID: 7 Object: window
Correct: 66 Percentage: 84.6154 ObjectID: 8 Object: counter
Correct: 67 Percentage: 85.8974 ObjectID: 9 Object: lighting
Correct: 69 Percentage: 88.4615 ObjectID: 10 Object: mirror
Correct: 73 Percentage: 93.5897 ObjectID: 11 Object: sink
Correct: 66 Percentage: 84.6154 ObjectID: 12 Object: ceiling
Correct: 70 Percentage: 89.7436 ObjectID: 13 Object: door
Correct: 60 Percentage: 76.9231 ObjectID: 14 Object: floor
Correct: 61 Percentage: 78.2051 ObjectID: 15 Object: picture
Correct: 66 Percentage: 84.6154 ObjectID: 16 Object: toilet
Correct: 73 Percentage: 93.5897 ObjectID: 17 Object: chair
Correct: 72 Percentage: 92.3077 ObjectID: 18 Object: stool
Correct: 71 Percentage: 91.0256 ObjectID: 19 Object: cushion
Correct: 59 Percentage: 75.641 ObjectID: 20 Object: void
Correct: 73 Percentage: 93.5897 ObjectID: 21 Object: bed
Correct: 69 Percentage: 88.4615 ObjectID: 22 Object: chest_of_drawers
Correct: 67 Percentage: 85.8974 ObjectID: 23 Object: table
Correct: 73 Percentage: 93.5897 ObjectID: 24 Object: tv_monitor
Correct: 62 Percentage: 79.4872 ObjectID: 25 Object: plant
Correct: 70 Percentage: 89.7436 ObjectID: 26 Object: shelving
Correct: 70 Percentage: 89.7436 ObjectID: 27 Object: appliances
Correct: 67 Percentage: 85.8974 ObjectID: 28 Object: misc
Correct: 71 Percentage: 91.0256 ObjectID: 29 Object: fireplace
Correct: 71 Percentage: 91.0256 ObjectID: 30 Object: sofa
Correct: 72 Percentage: 92.3077 ObjectID: 31 Object: column
Correct: 73 Percentage: 93.5897 ObjectID: 32 Object: beam
Correct: 70 Percentage: 89.7436 ObjectID: 33 Object: railing
Correct: 71 Percentage: 91.0256 ObjectID: 34 Object: stairs
Correct: 72 Percentage: 92.3077 ObjectID: 35 Object: seating
Correct: 77 Percentage: 98.7179 ObjectID: 36 Object: clothes
Correct: 77 Percentage: 98.7179 ObjectID: 37 Object: furniture
```

Die obige Ausgabe zeigt die accuracy per Object. Correct steht für wie viele der 78 Nullen bzw. Einsen des one-hot vectors korrekt gesetzt wurden. Percentage rechnet dies in Prozent um. Anschließend folgt die ObjectID und das zugehörige Objekt in Klartext. Am schlechtesten performen: floor, picture, void und plant. Eine Erklärung für floor und void könnte die größe dieser Objekte sein. Dadurch sind sie mit sehr vielen verschiedenen Objekten in kontakt und haben sehr viele unterschiedliche lösungen. Dies macht es dem Lernalgorithmus unter Umständen schwerer ein eindeutiges Muster zu erkennen.

2.6. Verbesserugsmöglichkeiten

Es könnte sinnvoll sein die accuracy des Netzes auf eine ander Art zu bestimmen. Es gibt insgesamt 78 Label, für ein Eingabeobjekt sind jedoch meist nur wenige dieser Label zutreffend. Wenn für ein Objekt zu dem nur ein einziges Label gehört genau dieses nicht vom Netz gesetzt wird ist der Hamming Score immer noch sehr gut, wenn alle anderen Label auch (richtig) nicht gesetzt wurden. Generell werden mehr Nullen als Einsen gesetzt, weshalb eine Idee zur Verbesserung wäre die richtigen Einsen und somit richtig erkannten Label bei der Berechnung der accuracy stärker zu gewichten als richtig gesetzte Nullen. Ansonsten gelten alle bereits erwähnten verbesserungsvorschläge aus Aufgabe 1 auch hier.

Integration TextImager

Die in Aufgabe 1 und Aufgabe 2 erstellten Netze sollten beide in den TextImager integriert werden.

Ziel ist es aus einem Satz alle Wörter zu erkennen, mit denen unser Netzwerk trainiert wurde und diese dann als Eingabe an die Netze zu übergeben. Anhand der Ausgabe sollen dann QSLinks erstellt werden.

Aufgrund von Problemen beim import des CoreNLPSegmenter sind wir auf den BreakIteratorSegmenter umgestiegen. Hierbei gab es jedoch bis zuletzt auch Probleme, die wir, auch nach längerer Rücksprache mit der Praktikumsleitung und anderen Arbeitsgruppen, nicht lösen konnten.

```
C:\Users\Besitzer\IdeaProjects\testproject\src\Main.java:35:21
java: cannot access de.tudarmstadt.ukp.dkpro.core.api.segmentation.SegmenterBase
  class file for de.tudarmstadt.ukp.dkpro.core.api.segmentation.SegmenterBase not found
```

Auch das Ausführen unserer Python-Programme über Java hat unter anderem aufgrund des xml.etree.ElementTree imports nicht funktioniert und wir haben auch hier, trotz Rücksprache, für diese Problem keine Lösung gefunden.

Als Alternative haben wir für beide Netze ein weiteres Python Programm geschrieben, in dem unsere trainierten Wörter aus einem Satz erkannt werden und dem Netz als passende Eingabe übergeben werden. Anhand der Ausgabe des Netzes wird dann eine xml-Datei erstellt, in der die so generierten QSLinks gespeichert werden.

Implementierung

Zunächst wird in beiden Programmen mit Hilfe von spacy der gegebene Satz in token aufgeteilt. Das Programm enthält eine Liste namens trained_objects, in der

alle im Training vorkommenden Objekte gespeichert sind. Die token werden mit den Elementen dieser Liste abgeglichen und wenn sie in der Liste vorhanden sind werden sie als Eingabe für das Netz genutzt.

Für A1_text.py werden nun also zwei Wörter, die figure und ground darstellen, als Eingabe an das trainierte Netz aus Aufgabe 1 übergeben.

Für A2_text.py wird ein Wort als Eingabe an das trainierte Netz aus Aufgabe 2 übergeben.

In beiden Programmen werden anschließend die Ergebnisse des Netzes als QSLink in einer xml-Datei gespeichert. Die xml-Dateien für die beiden Beispielsätze sehen wie folgt aus:

```
<?xml version="1.0"?>
<ROOT>
    <QSLINK figure="table" ground="chair" relType="PO"/>
</ROOT>
```

```
<?xml version="1.0"?>
<ROOT>
    <QSLINK figure="bathtub" ground="bathtub" relType="PO"/>
    <QSLINK figure="bathtub" ground="wall" relType="NTPP"/>
    <QSLINK figure="bathtub" ground="ceiling" relType="PO"/>
    <QSLINK figure="bathtub" ground="floor" relType="PO"/>
    <QSLINK figure="bathtub" ground="wall" relType="PO"/>
    <QSLINK figure="bathtub" ground="shower" relType="NTPP"/>
    <QSLINK figure="bathtub" ground="cabinet" relType="PO"/>
    <QSLINK figure="bathtub" ground="shower" relType="PO"/>
    <QSLINK figure="bathtub" ground="door" relType="PO"/>
    <QSLINK figure="bathtub" ground="toilet" relType="PO"/>
    <QSLINK figure="bathtub" ground="curtain" relType="PO"/>
    <QSLINK figure="bathtub" ground="picture" relType="EC"/>
    <QSLINK figure="bathtub" ground="plant" relType="PO"/>
    <QSLINK figure="bathtub" ground="curtain" relType="NTPP"/>
</ROOT>
```

Anwenderdokumentation

Das Programm welches zu dem Netz aus Aufgabe 1 gehört ist A1_text.py. Um dieses auszuführen müssen matterport3dhouse.xml und labelNet.pt (das trainierte Netz aus Aufgabe 1) im gleichen Ordner wie A1_text.py liegen. Außerdem sollte die requirements.txt installiert werden, falls dies noch nicht aus vorherigen Teilen geschehen ist.

Der zu untersuchende Satz kann in Zeile 15 des Codes (doc = nlp("The old book is on the chair next to the table.")) eingegeben werden. Die Ausgabe des Programms wird in einer vom Programm erzeugten xml-Datei namens file.xml gespeichert.

Dementsprechend gehört zu dem Netz aus Aufgabe 2 A2_text.py. Um dieses auszüführen müssen matterport3dhouse.xml und classifierNet.pt (das trainierte Netz

aus Aufgabe 2) im gelichen Ordner wie A2_text.py liegen Außerdem sollte die requirements.txt installiert werden, falls dies noch nicht aus vorherigen Teilen geschehen ist.

Der zu untersuchende Satz kann in Zeile 15 des Codes (doc = nlp("The old book is on the bathtub")) eingegeben werden. Die Ausgabe des Programms wird in einer vom Programm erzeugten xml-Datei namens file1.xml gespeichert.

Verbesserungsmöglichkeiten

Derzeit funktioniert das Programm für das Netz aus Aufgabe 1 nur dann, wenn genau zwei der im Training vorgekommen Wörter in dem gegebenem Beispielsatz vorkommen. Das Programm für das Netz aus Aufgabe 2 funktioniert nur, wenn genau ein Wort der im Training vorkommenden Wörter im Beispielsatz vorkommt.

Dies kann noch verbessert werden, sodass für Aufgabe 1 zum Beispiel alle bekannten Wörter im Beispielsatz herausgesucht werden und alle daraus möglichen Paare an Objekten als Eingabe an das Netz übergeben werden. So können auch QSLinks für ganze Texte generiert werden. Hierbei muss dann noch die Erstellung der xml Datei angepasst werden.

Für Aufgabe 2 könnte man statt genau einem Wort jedes bekannte Wort aus dem Satz als Eingabe für das Netz nutzen. Auch hier muss dann noch die Erstellung der xml Datei angepasst werden.

Literatur

- https://www.kaggle.com/tauseef6462/simple-feedforward-neural-network-using-pytorch
- https://pytorch.org/tutorials/beginner/blitz/neural_networks_tutorial.html
- https://www.kaggle.com/schmiddey/multiclass-classification-with-pytorch
- https://visualstudiomagazine.com/Articles/2020/12/15/pytorch-network.aspx?Page=1
- https://users.ics.aalto.fi/jesse/talks/Multilabel-Part01.pdf
- https://discuss.pytorch.org/t/multi-label-classification-in-pytorch/905
- https://stackoverflow.com/questions/42479902/how-does-the-view-method-work-in-pytorch
- https://www.linkedin.com/pulse/hamming-score-multi-label-classification-chandra-sharat
- https://stackoverflow.com/questions/37615544/f1-score-per-class-for-multi-class-classification
- https://discuss.pytorch.org/t/what-kind-of-loss-is-better-to-use-in-multilabel-classification/32203/23