Deep Learning a nyelvtechnológiában

Dávid Patrik

Konzulens: Ács judit

Tartalomjegyzék

1. A félévre kitűzött feladat

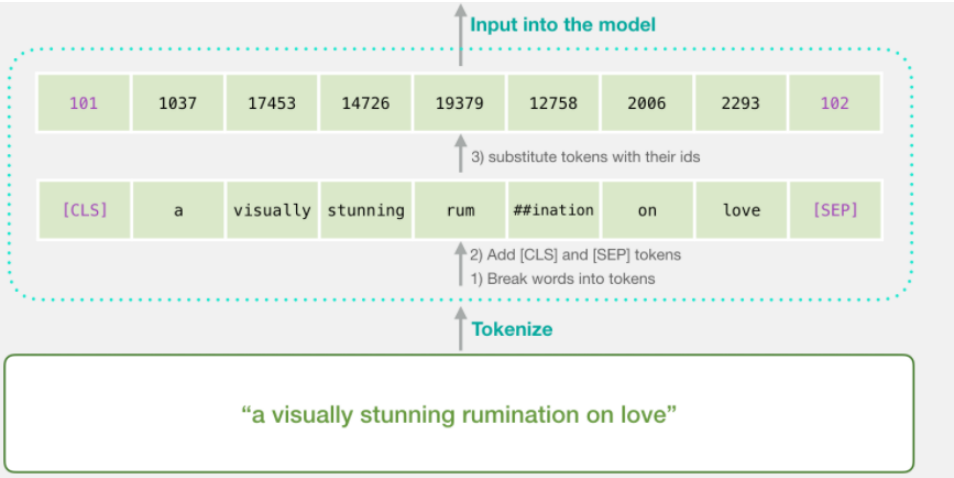
A félévre a BERT nevű technológia megismerését tűztük ki célul, továbbá a modell felhasználásával egy feladat megoldását. A feladat megoldást követően pedig kutatást végezni, hogy különböző tanítási paraméterekkel hogyan teljesít a modell.

2. BERT

A BERT dokumentációja részletes és nagy segítséget nyújtott a felépítésének a megismerésére, és a használatának az elsajátítására. Ezt a dokumentációt a `https://huggingface.co/transformers/` címen érhetjük el. Itt fellelhetők a huggingface további modelljei is.

2.1. Tokenizer

A BERT modell használhatához először meg kell értenünk mit is vár el bemenetnek a modellünk. A modell bementként tokeneket vár (vagyis azoknak az id-jét). A szövegből tokeneket a BERT tokenizer csinál számunkra. A félévben előre betanított tokenizert használtam, de külön feladatokhoz is taníthatunk tokenizert ha a feladat úgy kívánja. Én a feladatomban a base-multilingual-cased, előre betanított BERT modell tokenizerét használtam. A modell számomra legfontosabb tulajdonsága, hogy többnyelvű, így használhatom a következőkben részletezett feladatommal, ahol magyar mondatokkal dolgozom.



1. ábra: BERT tokenizer működési elve

A BERT tokenizer a tanítása során létrehozott egy szótárt, amibe a hasznosnak ítélt tokenek vannak. Egy token egy szó vagy szótöredék lehet, ami a szöveg kontextusából sok információt hordoz. Az ábrán látható például a "stunning" szó egy token maradt, de a "rumination" szót kettő szótöredékre tagolta. A tokenizer továbbá használ speciális tokeneket, mint [CLS] vagy [SEP], amik a modell számára adnak jelzéseket. Ezek alapértelmezetten be vannak kapcsolva, de ha a feladat úgy kívánja az `add_special_tokents` paraméter false állapotba állításával kikacsolhatjuk.

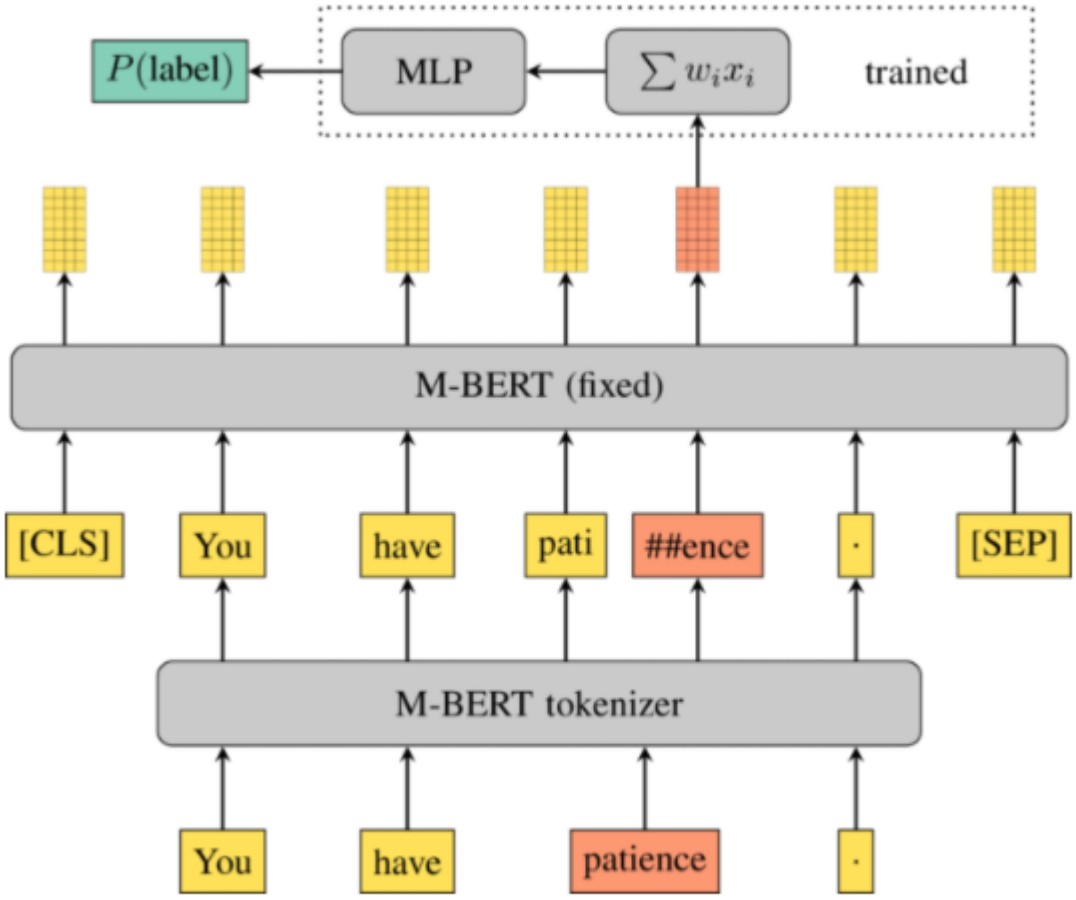
2.2. BERT modell

A BERT modell egy 12 rejtett réteget tartalmazó mélyneurális háló. Bemenetként token id vektort kap, a kimenete pedig egy tuple, amiben van egy `last_hidden_state` nevű tag, ami az utolsó rejtett réteg kimenetét adja meg, egy `pooler_output`, ami a félév keretein kívül esett, és egy `hidden_states` tag, amibe indexelve az egyes rejtett rétegek értékeit kapjuk meg. Az utóbbi csak a `output_hidden_states=True` paraméter beállítása esetén elérhető. Mindezeket összevetve a következő eredményt kapjuk. Továbbá látható, hogy hogyan olvashatók ki a rejtett rétegekből több mondat esetén mondatonként és a tokeneként a kapott mátrixok. Ezen felül fontos meghívni, ha nem szeretnénk tanítani a modell, a BERT model eval függvényét, ezzel azt jelezve annak, hogy nem tanítani szeretnénk, így olyan paramétereket nem ment el amik csak tanításhoz kellenek, nem pazarolva ezzel a rendszer memóriáját.

```
tokenizer = AutoTokenizer.from_pretrained("bert-base-multilingual-cased")
BertModel = AutoModel.from_pretrained("bert-base-multilingual-cased",s
                                                    output_hidden_states=True,
                                                    return_dict=True)


BertModel.eval()



input = self.tokenizer("Example sentence.", padding=True,
                                truncation=True, return_tensors="pt")
output = self.BertModel(**input)
output.hidden_states[INDEX_OF_LAYER][INDEX_OF_SENTENCE][INDEX_OF_TOKEN]
```

3. BERT segítségével megoldott feladat

A félév során a BERT-et felhasználtuk egy feladat megoldására. A feladat, noha jóval egyszerűbb, mint a modell segítségével megoldható problémák, a félév kereteibe, a modell használatának megértéséhez elég. A feladat maga egy program írása, ami bemeneteként egy magyar nyelvű mondatot kap, és egy sorszámot, ami azt jelzi, hogy hányadik szónak vagyunk kíváncsiak a nyelvtani esetére.



2. ábra: Feladatmegoldás sémája

A feladat megoldási sémáját a fenti képen láthatjuk. A mondatot először a tokenizeren visszük keresztül, majd a tokeneket a korábban tanult módon a modellbe töltjük, majd az általunk kiválasztott token mátrixát egy neurális hálóba töltjük, amit betaníthatunk a feladatunkra. Későbbiekben láthatjuk, hogy milyen hatással

van az eredményre, hogy a szónak mely tokenjét használjuk.

3.1. Adatok előkészítése

A neurális hálók által megoldott feladatok egyik nagy része a tanító és teszt adatok betöltése. Az én esetemben adatbázis már biztosítva volt, a feladatom mindössze ezeket pythonban előkészíteni volt. Ezt a pandas segítségével tettem, ami képes a .tsv ájlok olvasására.

```
    def preparedata(self):
        train_tsv = pd.read_csv('path', na_filter=None, quoting=3, sep="\t")
        dev_tsv = pd.read_csv('path', na_filter=None, quoting=3, sep="\t")
        training_data = []
        test_data = []


        for i, obj in enumerate(train_tsv.values, 0):
            input = self.tokenizer(obj[0], padding=True,
                                   truncation=True, return_tensors="pt")
            output = self.BertModel(**input)
            training_data.append([])

            training_data[i].append(output.hidden_states[self.Bertlayernumber][0][self.
    gettokennumber(obj[0], obj[2])].detach())
            training_data[i].append(self.casetonumber(obj[3]))

        for i, obj in enumerate(dev_tsv.values):
            input = self.tokenizer(obj[0], padding=True,
                                   truncation=True, return_tensors="pt")
            output = self.BertModel(**input)
            test_data.append([])
            test_data[i].append(output.hidden_states[self.Bertlayernumber][0][self.
    gettokennumber(obj[0], obj[2])].detach())
            test_data[i].append(self.casetonumber(obj[3]))

        self.trainloader = torch.utils.data.DataLoader(training_data, batch_size=self.
    BatchSize, shuffle=True)
        self.testloader = torch.utils.data.DataLoader(test_data, batch_size=self.BatchSize,
    shuffle=True)
```

A függvény első fele a fájlok olvasása pandas segítségével. A második fele az adatok összerakása tanuláskor iterálható formátumba for ciklus segítségével. Ezen a megoldáson sokat javítana, ha nem egyesével vinném

a mondatokat keresztül a BERT modellen, hanem a rendelkezésre álló memória alapján egyszerre akár 64-et vagy többet is. Ezt idő hiányában nem tudtam megoldani, de sokat gyorsítana a program futásán. A pandas objektumokon a .values attribútum segítségével iterálhatunk végig. A gettokennumber nevű függvény megadja, hogy a mondatban a sorszám alapján megadott szónak, az utolsó tokenje a tokenek között hányadik. A casetonumber függvény integerré konvertálja a latin rövidítéssel megadott nyelvtani eseteket.

3.2. Neurális háló

A feladat megoldására a konzulensem ajánlatára, egy egyszerű lineáris neurális hálót használtam. Az oka ennek az, hogy a feladat oroszlán részét a BERT modell végzi, ez a neurális háló már csak feladatspecifikus teendőket lát el.

```
class SimpleClassifier(nn.Module):
    def __init__(self, input_dim, output_dim, hidden_dim):
        super().__init__()
        self.input_layer = nn.Linear(input_dim, hidden_dim)
        self.relu = nn.ReLU()
        self.output_layer = nn.Linear(hidden_dim, output_dim)

    def forward(self, X):
        h = self.input_layer(X)
        h = self.relu(h)
        out = self.output_layer(h)
        return out

Net = SimpleClassifier(
            input_dim=768,
            output_dim=6,
            hidden_dim=50
        )
```

A háló feladathoz kötött paraméterei a bemeneti dimenzió és a kimeneti dimenzió, ami 768, mivel a BERT kimenete annyi elemű, és 6 mivel annyi osztályunk van. A rejtett dimenzió állítható paramétere a hálónak.

4. Tanítás

A tanításhoz először definiáltunk loss függvényt, én a CrossEntropyLoss-t használtam, optimizert amihez SGD optimizert használtam, de Adam optimizer is megfelelő, és schedulert. Ezek után a tanító és eval függvény a következőképpen néz ki.

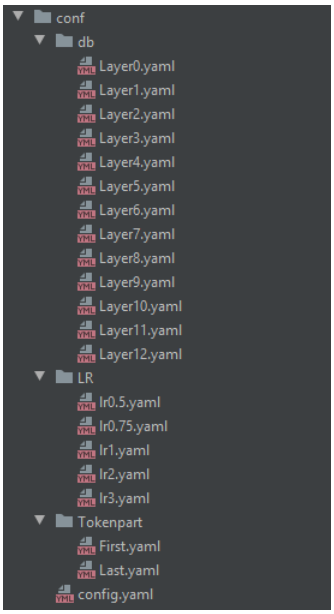
```
    def train(self, epoch):
        self.Net.train()
        running_loss = 0.0
        correct = 0.0
        total = 0

        for i, data in enumerate(self.trainloader, 0):
            tensors, labels = data

            tensors = tensors.cuda()
            labels = labels.cuda()

            self.optimizer.zero_grad()
            outputs = self.Net(tensors)
            loss = self.criterion(outputs, labels)
            loss.backward()
            self.optimizer.step()

            with torch.no_grad():
                running_loss += loss.item()
                _, predicted = torch.max(outputs, 1)
                correct += predicted.eq(labels).sum().item()
                total += labels.size(0)

        tr_loss = running_loss / i
        tr_corr = correct / total * 100
        print("Train epoch " + str(epoch + 1) + "  correct: " + str(tr_corr))
        return tr_loss, tr_corr


    def val(self, epoch):
        self.Net.eval()
        running_loss = 0.0
        correct = 0.0
        total = 0

        for i, data in enumerate(self.testloader, 0):
            tensors, labels = data

            tensors = tensors.cuda()
            labels = labels.cuda()
```

```
42
43              with torch.no_grad():
44                  outputs = self.Net(tensors)
45                  loss = self.criterion(outputs, labels)
46                  running_loss += loss.item()
47                  _, predicted = torch.max(outputs, 1)
48                  correct += predicted.eq(labels).sum().item()
49                  total += labels.size(0)
50
51          val_loss = running_loss / i
52          val_corr = correct / total * 100
53          print("Test epoch " + str(epoch + 1) + " loss: " + str(running_loss / i) + " correct
    : " + str(val_corr))
54          return val_loss, val_corr
```

Az programot egy classban írtam meg, emiatt szükségesek a self paraméterek.

5. Paraméterek optimalizálása Hydra segítségével

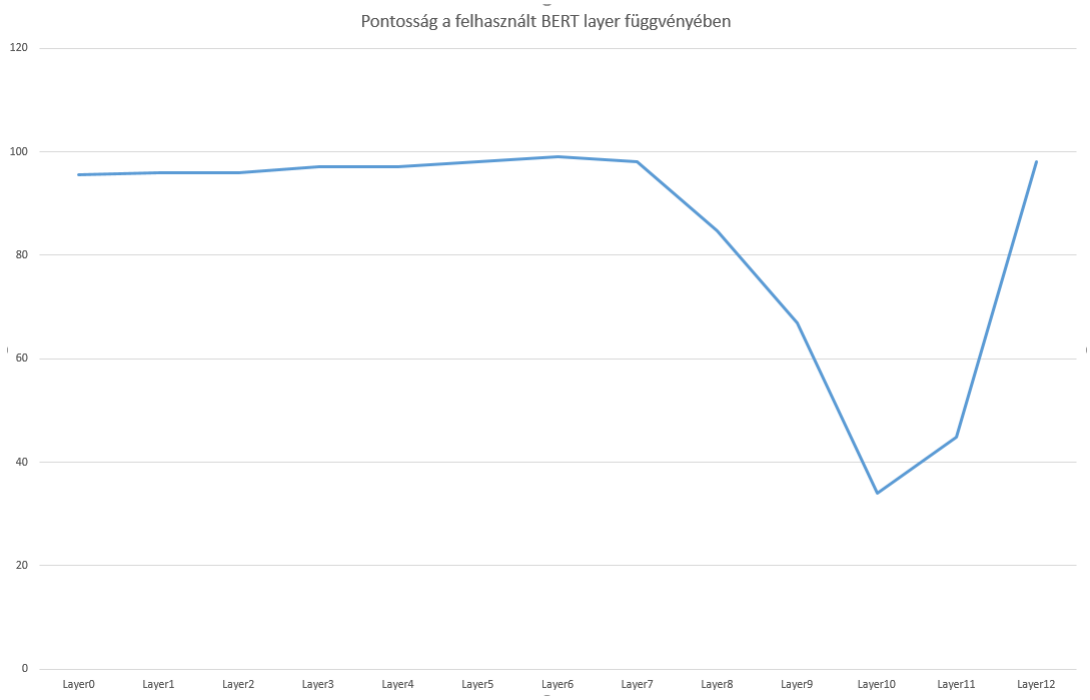
A Hydra egy python tool, ami kitűnően használható egy függvény sok egymás utáni lefuttatására úgy, hogy a program többi részét nem szükséges lefuttatnunk. A használatához csinálnunk kell config fájlokat .yaml formátumban ezeket mappák segítségével egy groupba rendezhetjük. Ezek után lefuttathatjuk a programunkat parancssorból és a -m parancs segítségével multirunt indíthatunk, például: `python main.py -m Tokenpart=Last LR=lr0.5,lr0.75,lr1,lr2,lr3 db=Layer6,Layer12,Layer13`. Ez a futás a következő fájl rendszerből választ configurációkat.



3. ábra: Konfiguráció

A hydra segítségével a feladatot lefuttattam a BERT minden egyes rejtett layer-ére, első és utolsó token alapú nyelvi eset vizsgálatra, és az SGD optimizer learning rate variálására. A hydrával való futtatást a @hydra.main() függvénnyel jelezzük, itt megadhatjuk a config fájlok nevét és elérési útját.

```
import hydra
...

@hydra.main(config_path="conf", config_name="config")
def BertMainFunction(cfg: DictConfig):
    param.configure(learningrate=cfg.LR.learningrate, Tokenpart=cfg.Tokenpart.Token,
    Bertlayernumber=cfg.db.Bertlayer)
    train_accs = []
    train_losses = []
    ...

```



4. ábra: Eredmények különböző rétegekre

A rétegek közül a legtöbb réteg jól teljesített, a legjobbnak a 6. réteg bizonyult, a 8, 9, 10 illetve 11. rétegek a feladat szempontjából nem adtak használható megoldást. Az első token használata a magyar nyelvben nem szintén rossz megoldásokhoz vezetett. Ez betudható a magyar nyelv felépítésének. A learning rate értékére a legjobb eredmény a 0.5e-1 értéknél lett, itt az érték nem volt túl , hogy beragadjon a modell rossz értékekre. További tesztek szükségesek, hogy alacsonyabb érték is felvehető-e.

6. Jövőbeli tervek

A félév során szerzett ismereteimet szeretném az elkövetkezendő félévek során elmélyíteni a Diplomaterv 1-2 keretein belül.