Guide TP détection d'anomalies avec auto-encodeurs

4A ESIEA

Alexandre Dey

Année 2020-2021

Airbus CyberSecurity, IRISA

Introduction

Etape 1 : Préparatifs

Etape 2 : Entrainer un auto-encodeur

Etape 3 : interpréter les résultats

Etape 4 : Résultats

Introduction

Introduction - Déroulement

Préparation des données

- Quelque soit le problème, on ne peut espérer le résoudre qu'en le comprenant. En data science, cette compréhension est souvent acquise grâce à l'exploration des données
- Cette exploration nous permet d'identifier à la fois les algorithmes que l'on va pouvoir utiliser, mais aussi les pré-traitements à appliquer à la données pour obtenir les meilleures résultats avec ces algorithmes
- En data science, il est important que les pré-traitements que l'on applique à la donnée soient reproductibles

Introduction - Déroulement

Construire l'auto-encodeur et l'entraîner

- Les réseaux de neurones peuvent traiter plusieurs type de variables (ici, numérique, binaires et catégorielles)
- Nos données contiennent des attributs hétérogènes (différents types). Chaque attribut aura donc deux parties dédiées dans le réseau, une en charge "d'uniformiser" la donnée pour le reste du réseau (les XyzEncoder), et une autre chargée de reconstituer la valeur de l'attribut (les XyzDecoder)
- Entre les deux, le cœur du réseau s'occupe lui d'identifier quels attributs vont fréquemment ensemble pour compresser au mieux la donnée (en perdant un minimum d'information)
- Pour la détection d'anomalie, on entraîne le réseau sur les données normales, et on se sert de l'erreur de reproduction pour trouver des anomalies

Introduction - Déroulement

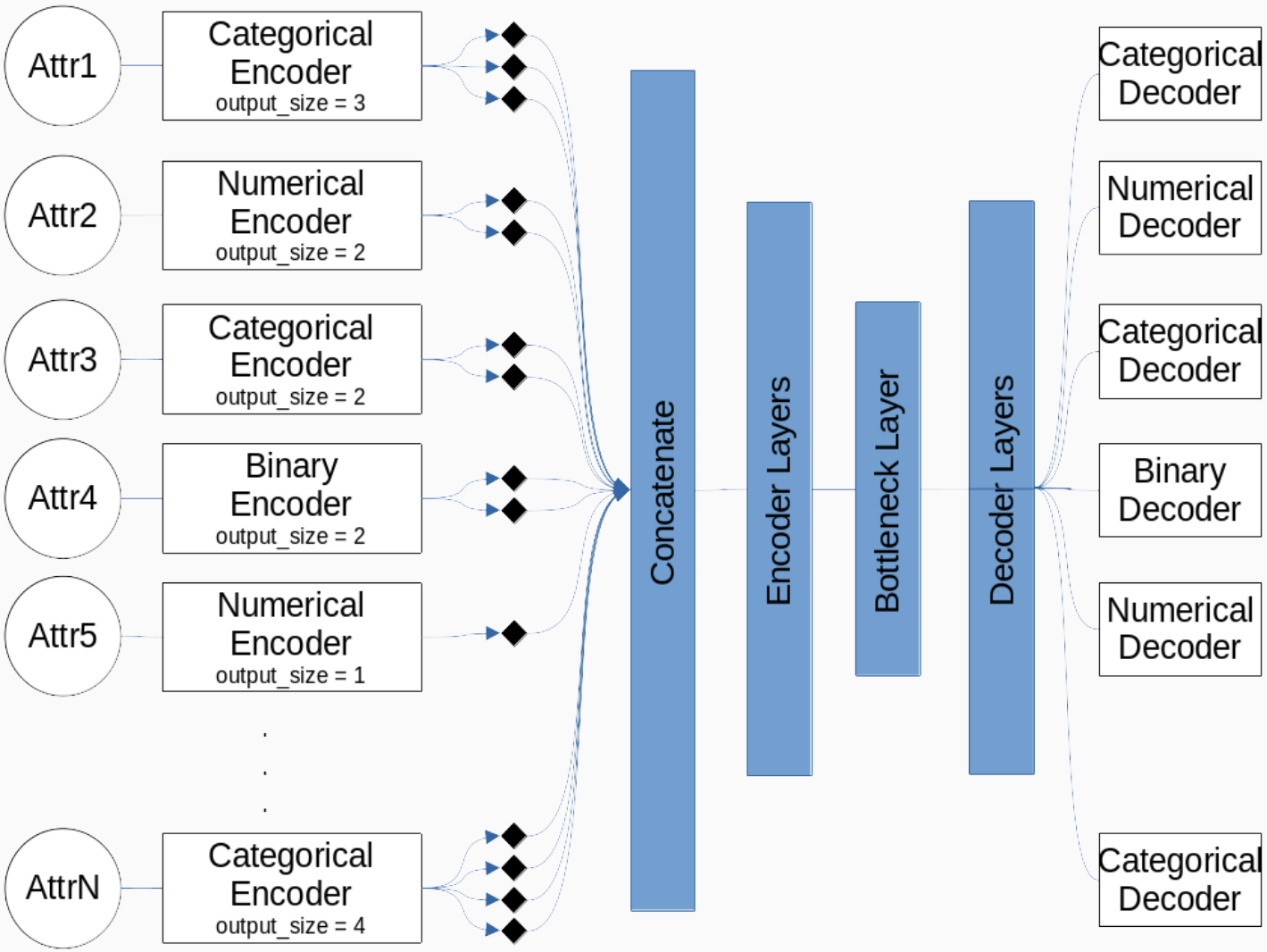

Figure 1 – Schéma de la structure d'auto-encodeur

Introduction - Déroulement

Calibrer et donner un score d'anomalie

- À ce stade, le réseau va donner l'erreur de reproduction par attribut pour chaque élément, ce qui n'est pas facilement interprétable
- Une méthode simple consiste à faire la somme de toutes les erreurs, mais elle est assez limitée. En effet, certains attributs sont plus difficiles que d'autres à répliquer, même pour les données normales (ex : forte variance). Ajouté à ça les différentes méthodes employées pour le calcul de chaque erreur, et il devient recommandable d'uniformiser ces erreurs (similaire à ce que l'on fait pendant le pré-traitement des données) avant de les combiner

Interpréter le score

- En théorie, plus le score calculé précédemment sera élevé, plus il sera probable que l'élément soit quelque chose que l'on souhaite détecter (en cybersécurité, une attaque)
- En pratique, il existe des anomalies qui ne sont en rien liées à ce que l'on cherche à détecter (des faux positifs).
- Le problème est le suivant : si le seuil est trop bas, il y aura beaucoup de faux positifs à traiter (souvent par un humain), et s'il est trop haut, on va louper beaucoup de choses pourtant intéressantes
- On cherche donc un juste milieu, grâce à quelques exemples de ce que l'on cherche à détecter

Etape 1 : Préparatifs

Etape 1 - Identifier les types de variables

Numériques

Ces variables sont des entiers ou des nombres décimaux, et constitue la majorité des types d'attributs de ce dataset

Binaire

Des entiers qui ne peuvent (normalement) être que 0 ou 1

Catégorielles

Ce sont, en l'occurrence, des chaînes de caractères

- Ne pas utiliser l'attribut "attack_cat" car vous ne trouverez pas cette variables dans "hidden.csv"
- Vous pouvez utiliser la méthode .describe() des dataframes pour obtenir des infos sur les différentes colonnes du dataset
- Vous avez aussi accès, dans "UNSW-NB15_features.csv", à une description de chacun des champs

Etape 1 - Remplir le dictionnaire conf

```
###### Replace [...] with a list of all categorical attributes
for attr in ["proto"]:
    conf["attributes"][attr] = "categorical"

###### Replace [...] with a list of all numerical attributes
for attr in ["sload", "dload"]:
    conf["attributes"][attr] = "numerical"

###### Replace [...] with a list of all binary attributes
for attr in ["is_something_something"]:
    conf["attributes"][attr] = "binary"
```

Etape 1 - Transformation des variables

Numériques

Plusieurs méthodes possibles, plus ou moins efficaces. À vous de tester. Ex : Standarisation, diviser par le max, diviser pas le 99ème centile, utiliser des logarithmes, ...

Binaire

Revient surtout à corriger le bruit (les valeurs différentes de 1)

Catégorielles

Déjà implémenté

- Dans l'étape 1, pas besoin d'appeler transform
- Transformation nécessaire pour les variables catégorielles
- Très importants pour les variables numériques (cf. slide d'après) pour permettre au réseau d'apprendre correctement

Etape 1 - Transformation des variables

```
transformed = pd.DataFrame()
durFE = libTP.feature_engineering.NumericalFE()
transformed["dur"] = durFE.fit_transform(train_df["dur"])
dloadFE = libTP.feature_engineering.NumericalFE()
transformed["dload"] = dloadFE.fit_transform(train_df["dload"])
```

```
train_df[["dur", "dload"]].head(10)
```

	dur	dload
0	1.155112	6.651476e+04
1	0.001119	6.362824e+05
2	0.333782	6.878741e+04
3	0.000005	0.000000e+00
4	0.038795	6.805001e+05
5	0.000000	0.000000e+00
6	0.000009	0.000000e+00
7	2.024546	3.267893e+03
8	2.058771	6.034668e+03
9	0.034907	1.647807e+07

```
transformed[["dur", "dload"]].head(10)
```

	dur	dload
0	0.239797	0.668885
1	0.000349	0.804901
2	0.089948	0.670909
3	0.000002	0.000000
4	0.011887	0.808948
5	0.000000	0.000000
6	0.000003	0.000000
7	0.345642	0.487408
8	0.349156	0.524344
9	0.010716	1.000904

Figure 2 – Pour le jeu de données brut, les valeurs sont entre 0 et 2 pour *dur* et entre 0 et 10e7 pour *dload*. Une fois transformé, les deux sont proches de [0-1]

Etape 1 - Exemple de la standardisation

```python
class NumericalFE(BaseFE):
    def __init__(self):
        super().__init__()
        self.mu = 0
        self.sigma = 0
    def fit(self, data):
        self.mu = data.mean()
        self.sigma = data.std()
    def transform(self, data):
        # .astype(np.float32) est utile car pytorch demande des
        # Float et non des double (float64) par défaut pour numpy
        return ((data - self.mu)/self.sigma).astype(np.float32)
```

Etape 1 - Créer le dictionnaire transforms

```
transforms = {}
for attr, t in conf["attributes"].items():
    # On instancie un XyzFE en fonction du type de l'attribut
    if t == "categorical":
        transforms[attr] = libTP.feature_engineering.CategoricalFE()
    if t == "numerical":
        transforms[attr] = libTP.feature_engineering.NumericalFE()
    if t == "binary":
        transforms[attr] = libTP.feature_engineering.BinaryFE()
    # On trouve les bons paramètres grâce à fit et aux données
    transforms[attr].fit(dataframe[attr])
```

Etape 1 - Structure du réseau

Ici, il faut donner les paramètres qui seront nécessaires au moment de la création du réseau

```
for attr, t in conf["attributes"].items():
    conf["network"][attr] = {"type": t, "output_size": 2}
    if t == "categorical":
        conf["network"][attr]["voc_size"] = transforms[attr].max_size
        # Les variables catégorielles peuvent profiter d'avoir une
        # output_size plus élevée
        conf["network"][attr]["output_size"] = 3
    if attr == "proto":
        # On peut aussi choisir un paramètre spécifique par attribut
        conf["network"][attr]["output_size"] = 2
```

Etape 2 : Entrainer un auto-encodeur

Etape 2 - Implémenter la structure du réseau

Les XyzEncoder

Correspondent aux toutes premières couches du réseau, qui traitent directement les entrées

Les XyzDecoder

Correspondent aux couches de sortie. Il faudra changer le nombre de neurones et la fonction d'activation en fonction du type d'attribut

Les XyzLoss

Mesure l'erreur de reproduction de l'auto-encodeur en comparant la valeur attendue et celle obtenue.

Etape 2 - Binary et NumericalEncoder

- Très similaire à ce qui a été fait dans le TD
- ATTENTION : Vous n'avez qu'une valeur d'entrée car un seul attribut est traité par chaque XyzEncoder
- Vous êtes libre du choix de la fonction d'activation

```
class NumericalEncoder(torch.nn.Module):
    def __init__(self, params):
        super().__init__()
        self.layer = Sequential(
            Linear(1, params["output_size"]),
            Sigmoid()
        )
    def forward(self, x):
        return self.layer(x)
```

Etape 2 - CategoricalEncoder

- Des entiers correspondant à des catégories en entrée
- Il faut utiliser torch.nn.Embedding
- Subtilité : Embedding est fait pour traiter des séquences et retourne une séquence contenant 1 élément. On peut supprimer cette dimension inutile avec .squeeze()

```
class CategoricalEncoder(torch.nn.Module):
    def __init__(self, params):
        super().__init__()
        self.layer = Sequential(
            # nb_cat, max_size, voc_size, ... vous donnez le nom
            # que vous voulez
            Embedding(params["nb_cat"], params["output_size"]),
            Sigmoid()
        )
    def forward(self, x):
        return self.layer(x).squeeze()
```

Etape 2 - Les XyzDecoder

Binary

Très similaire au TD, peut être considéré comme une classification binaire

Numerical

Similaire à Binary, mais attention à la fonction d'activation, Sigmoid ne retourne des valeurs qu'entre 0 et 1, Tanh entre -1 et 1, ReLU > 0. Fonction d'activation à choisir en fonction de la méthode de transformation

Categorical

Ici on est dans un cas de classification multiclasse : il faut autant de valeurs de sortie qu'il y a de catégories, et utiliser une fonction d'activation permettant d'exprimer la probabilité d'être une classe ou une autre (softmax)

Etape 2 - Les XyzLoss

Binary

Pareil que pour le TD

Numerical

Chercher dans le cours et dans la doc de pytorch une loss adaptée pour la régression en présence d'anomalies

Categorical

Chercher dans la doc de pytorch une loss adaptée pour la classification multiclasse

Etape 2 - La boucle d'entraînement

```
# Ce call back va arrêter l'apprentissage dès que 10 epochs
# seront passées sans une amélioration de 'val_loss' d'au moins 0.01
early_stopping = pl.callbacks.EarlyStopping(min_delta=0.01,
patience=10, monitor='val_loss')
# Celui-ci sauvegarde le modèle avec la meilleure val_loss
checkpoints = pl.callbacks.ModelCheckpoint(monitor="val_loss",
dirpath="conf/checkpoints/")
# On précise au trainer d'utiliser les callbacks et de s'arrêter au
# plus tard au bout de 1000 epoch
trainer = pl.Trainer(callbacks=[early_stopping, checkpoints],
max_epochs=1000)
trainer.fit(model, train_dataloader=train_loader,
val_dataloaders=validation_loader)
# Une fois entraîner on restore le meilleur modèle
model = MyNet.load_from_checkpoint(checkpoints.best_model_path)
```

Etape 2 - La calibration et le scoring

Méthode calibrate de autoencoder.py

- Vous avez accès aux valeurs de chaque loss (i.e. par attribut) et pour chaque élément des données d'entrées grâce au dictionaire 'losses'
- Vous pouvez vous en servir pour trouver les paramètres d'une transformation pour chaque loss

Example pour une standardisation

```
for attr in losses:
    self.scorer_parameters[attr] = {
        'mu': losses[attr].mean(),
        'sigma': losses[attr].std()
    }
```

Etape 2 - La calibration et le scoring

Méthode score de autoencoder.py

- Vous avez accès aux loss par attribut grâce à 'loss_dict'
- Vous devez y appliquer la fonction de transformation et stocker le résultat dans le dictionaire scaled_loss
- le score final d'un élément est la somme de scaled_loss pour cet élément et pour chaque attribut

Example pour une standardisation

```
for k in loss_dict:
  scaled_loss[k]=(loss_dict[k]-self.scorer_parameters[k]["mu"])/
                 self.scorer_parameters[k]["sigma"]
  # La standardisation renvoi des valeurs négatives quand
  # la valeur initiale est inférieur à la moyenne (i.e "normale")
  # Pour éviter d'atténuer le score final pour les anomalies,
  # on peut utiliser torch.clip (ici on borne entre 0 et 10)
  scaled_loss[k] = torch.clip(scaled_loss[k], 0, 10)
```

Etape 2 - Sauvegarder

Les poids du réseau

- Les poids du réseau sont déjà sauvegardés grâce au checkpoint
- Le chemin vers votre modèle sauvegardé est stocké dans le membre best_model_path de la classe ModelCheckpoint()
- NOTE : vous pouvez soit réutiliser ce fichier directement, soit le renommer et le déplacer pour plus de facilité au chargement

Les paramètres du scorer

- Les checkpoints ne contiennent pas scorer_parameters
- Vous pouvez utiliser la méthode save_scorer pour les sauvegarder
- Pour les recharger, vous pouvez utiliser load_scorer

Etape 3 : interpréter les résultats

Etape 3 - Début

Recharger la config et le modèle

- comme à l'étape précédente pour la conf
- pour le modèle, utiliser le checkpoint du meilleur modèle avec load_from_checkpoint
- load_scorer pour recharger les paramètres pour le scoring

Recharger la config et le modèle

Comme à l'étape précédente mais charger "evaluate.csv"

Appeler le modèle

raw_data["score"] = model(transfomed_data[:]).numpy()

- '[:]' est requis pour retourner tous les éléments contenus dans le dataset (et non juste la référence au dataset)
- .numpy() sert à convertir les tensor (rerpésentation pytorch) en array numpy (compatibles avec les dataframe pandas)

Etape 3 - Trouver le seuil de détection

Objectif

Trouver la valeur pour le score au delà de laquelle on va considérer les éléments comme anormaux. L'objectif étant de maximiser la qualité de la détection

> Il faut identifier la bonne métrique pour de la détection d'anomalie (cf cours). Le meilleur seuil de détection la maximise

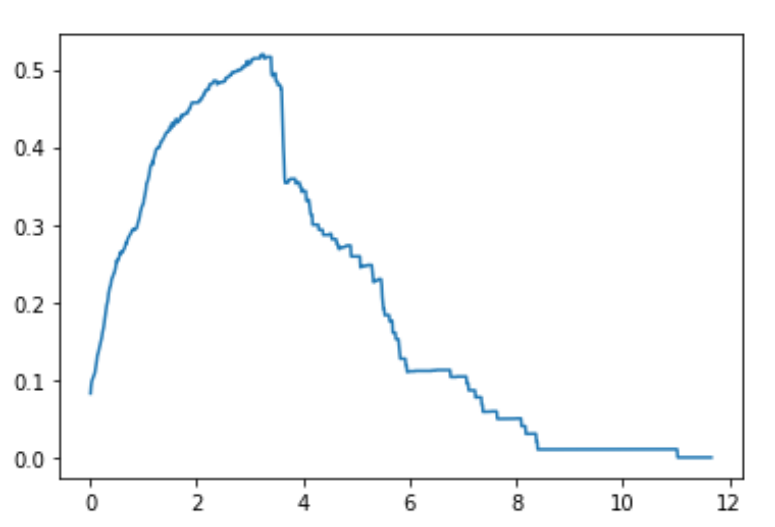

Figure 3 – Exemple de courbe à inclure dans le rapport

Etape 4 : Résultats

Etape 4 - Améliorer les performances

Ce qui améliorera les performances, dans l'ordre d'importance (liste non exhaustive) :

1. Modifier la fonction de scaling dans .calibrate et .score
2. Modifier la méthode de trasnformation des variables numériques
3. Changer la structure du réseau (rajouter des couches, dropout, régularisation l1/l2, ...)

Etape 4 - Labéliser le dataset unknown

Bien respecter le format CSV avec trois colonnes

1. score : le score d'anomalie en sortie de l'autoencodeur
2. pred_label : le label prédit (1 anomalie, 0 normal)
3. hidden_label : fournit dans le jeux d'entrée