Theorie Rapport: Machine Learning

Auteur: Brian van den Berg

Introductie

Omdat de Neural Networks opdracht een vrije opdracht was, heb ik gekozen voor de eerste keuze uit de opdrachten. Ik heb een opdracht gedaan uit een wedstrijd op Kaggle. De opdracht waar ik me mee bezig gehouden heb is "Titanic - Machine Learning from Disaster". Over het algemeen zijn de meest behaalde resultaten voor deze dataset rond de 80%, dus dat is ook mijn doel.

Deze competitie functioneert normaliter als een introductie tot Kaggle competities en de meest gangbare manier om dit probleem op te lossen met de data die gegeven is, is door middel van een Random Tree/Forest Classifier, maar het is interessant om te kijken hoe een Neural Network hierin vergelijkt met een Random Tree/Forest.

Uit een eerste blik op de data lijkt het erop dat een groot deel van de opdracht, het voorbereiden van de data wordt. Er is vrij veel tekstdata en na het omzetten van tekst naar cijfers, is er veel categoriale data en numerieke data die moet worden genormaliseerd.

Inhoud

Introductie 1
Preprocessing 3
Missende Data 3
Onbruikbare kolommen 3
De kolom ‘Sex’ 3
De kolom ‘Embarked’ 3
Numerieke Data 3
Categoriale Data 4
Features (X) en Labels (Y) 4
Train en Test split 4
Balanceren van de Data 4
Neural Network 5
Model 5
Trainen 6
Batch Size 6
Validatie Set 7
Inzichten 7
Resultaten 8
Precision and Recall 8
F1-Score 8
Conclusie 9

Preprocessing

Een belangrijk deel van het maken van een Neural Network is het verzamelen van een grote kwantiteit van correlerende data. De dataset is gelukkig gegeven in de competitie en deze is ook bijgevoegd in de bijlagen van de DLO Submission. Maar in dit hoofdstuk zal ik beschrijven hoe de data is voorbereid voor het model. Hier is een voorbeeld van de data:

PassengerId	Survived	Pclass	Name	Sex	Age	SibSp	Parch	Ticket	Fare	Cabin	Embarked
1	0	3	Braund, Mr. Owen Harris	male	22.0	1	0	A/5 21171	7.2500	NaN	S

Missende Data

Als allereerste is het belangrijk om te checken of de data wel compleet dit, dat is zichtbaar in de afbeelding hiernaast. Zoals u kunt zien mist ~20% van de rijen een leeftijd, 77% een slaapvertrek en een verschrikkelijk kleine hoeveelheid mensen mist een locatie van vertrek. Leeftijd is een belangrijke feature om mee te nemen in het model, dus dat kan worden opgevuld met de gemiddelde leeftijd. ‘Embarked’ mist er zo weinig en heeft maar drie categorieën, dus die kan worden opgevuld met de mediaan. Als laatste mist ‘Cabin’ een grote hoeveelheid data en daarom laat ik deze kolom vallen. Daarnaast waren er mensen met meerdere slaapvertrekken. Ik heb geprobeerd om de verdieping van iemand zijn/haar slaapvertrek mee te nemen in het algoritme, maar daar leek geen verschil uit te komen en het bracht wel 8 extra kolommen aan ‘One Hot Encoded’ complexiteit aan de data.

```
Percentage of rows with NaN values for each column:
Survived      0.000000
Pclass        0.000000
Name          0.000000
Sex           0.000000
Age          19.865320
SibSp         0.000000
Parch         0.000000
Ticket        0.000000
Fare          0.000000
Cabin        77.104377
Embarked      0.224467
```

Onbruikbare kolommen

De kolommen die ik heb laten vallen zijn ‘Naam’, ‘Ticket’ en ‘Cabin’, omdat deze niks zeggen over het overleven van de mensen, te complex zijn en/of een combinatie van de twee.

De kolom ‘Sex’

In de data staat een kolom voor geslacht, maar dit is in de vorm van tekst. Een Neuraal netwerk kan tekst niet gebruiken, dus in deze vorm is het onbruikbaar, maar gelukkig is daar een oplossing voor. Toevallig is een Neuraal Netwerk namelijk wel heel erg goed in binaire data en male/female kan ook omgezet worden naar 0/1 en dat is dan ook wat er in mijn Notebook gedaan is.

De kolom ‘Embarked’

Mijn hypothese bij het gebruiken van deze feature was dat de plaats waar iemand instapte mogelijk een maat van status of rijkdom met zich mee kon brengen, daarom vond ik het belangrijk om te testen of deze feature van belang was bij het model. Achteraf klopte dat niet en om dezelfde reden als waarom ‘Cabin’ eruit gehaald is, heb ik nu ook ‘Embarked’ uit de data gehaald zonder dat het een impact lijkt te hebben op de score of balans van mijn model.

Numerieke Data

De numerieke data waren vrij makkelijk om voor te bereiden, want binnen deze data kunnen eerst de missende waardes bijgevuld worden met het gemiddelde en nadat dat gedaan is, is het een kwestie van de numerieke data normaliseren. Dit is gedaan door middel van een ‘Robust Scaler’ van sklearn. De rede waarom ik voor deze scaler heb gekozen, was omdat de data extreme waardes bevatte en de robust scaler kan deze waardes op een correcte manier representeren.

Categoriale Data

Dit is op het moment alleen nog maar de kolom 'Pclass', maar dient als een mooie manier om uit te leggen hoe hier mee om gegaan kan worden. Categoriale data staan vaak op zichzelf gerepresenteerd als een reeks aan waardes die meer informatie bevatten achter de waarde dan te zien is in de data. Zo is bijvoorbeeld bij 'Pclass' de klasse van de ticket die je gekocht hebt (1ste klas, 2de klas en 3de klas). Maar omdat er zo'n groot verschil is tussen de socio-economische status die bij de verschillende klassen hoort, is het representeren van deze data op een numerieke scale niet representatief over de data. Daarom bestaat One Hot Encoding.

De manier waarop One Hot Encoding werkt is dat er voor elke unieke waarde een kolom wordt aangemaakt en elke klasse waar een persoon niet in valt wordt op 0 gezet en alleen de kolom waar de persoon bij hoort staat op 1. In wijzen wordt de data dus omgezet van 'persoon X heeft als Pclass 1' naar 'persoon 1 is 1ste klas, persoon 1 is niet 2de klas en persoon 1 is ook niet 3de klas'. Door deze representatie van de data zal het neurale netwerk de waardes zien als drie aparte binaire features.

Features (X) en Labels (Y)

De labels zijn makkelijk om uit deze data op te halen. Namelijk, de kolom 'Survived' is al een binaire classificatie of een persoon de Titanic overleeft heeft. Deze kan dan ook geselecteerd worden en vervolgens uit de dataframe verwijdert worden.

De features zijn nu een combinatie van de dataframe en de one hot encoded categoriale data. De vorm van de matrix aan data en labels is te zien in de afbeelding hiernaast.

```
Labels (Y) shape: (891,)
Features (X) shape: (891, 7)
```

Train en Test split

In mijn uitwerking heb ik de normale 70/30 regel toegepast op de data voor een train/test split. De test data kan alleen beter gezien worden als validatie data, omdat de echte test data geen labels heeft en beoordeelt zou worden door Kaggle. Echter wil ik mijn model optimaliseren, dus ik heb test data nodig waar ik mijn model mee kan benchmarken. Dit was gedaan door de train_test_split functie van sklearn.

Balanceren van de Data

Uit het evalueren van mijn model kwam dat de recall en precision niet evenredig waren bij het trainen van mijn model. De rede waarom dat kan komen, is omdat er een slechte balans is in de data tussen features die behoren tot de overlevenden en de features die horen bij mensen die gestorven zijn. Om dit probleem op te lossen en rekening te houden met de schaarse hoeveelheid data, heb ik van de library 'imblearn' (inbalanced learning) de Over Sampler gebruikt. Deze voegt op basis van bestaande data nieuwe data toe om een gelijke distributie van klassen te krijgen in de labels. Deze methode is alleen geschikt voor tabel data en niet voor foto's of tekst, maar voor tabel data werkt dit perfect. Het is alleen belangrijk dat dit alleen op de training data wordt toegepast, want de validatie en testing set moeten de normale ongebalanceerde distributie van klassen weergeven.

Neural Network

Nu dat de data uitgelegd is en hoe ik het heb voorbereid voor mijn model, is het tijd om het Neurale Netwerk uit te leggen en wat de termen betekenen binnen het netwerk.

Model

```
# Create a neural network model using Keras
model = tf.keras.Sequential([
    tf.keras.layers.Dense(64, activation='relu', input_shape=(X_train.shape[1],)),
    tf.keras.layers.Dense(32, activation='relu'),
    tf.keras.layers.Dropout(0.5),
    tf.keras.layers.Dense(1, activation='sigmoid')
])

# Compile the model with a lower learning rate
model.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=0.001),
              loss='binary_crossentropy',
              metrics=['accuracy'])
```

Om te beginnen met mijn model zal ik de keuzen voor mijn laatste laag uitleggen, want dit is de meest belangrijke en deterministische laag in een Neuraal Netwerk. Als output voor mijn netwerk wil ik een classificatie of iemand het heeft overleeft of niet. Dit noem je een binaire classificatie en standaard gebruik je hier maar een enkel neuron voor. De activatie van deze neuron is ‘sigmoid’. Dit is een niet-lineaire functie die een waarde tussen 0 en 1 teruggeeft. Deze activatie is veel gebruikt in de laatste laag van een binair classificatie model, omdat het de kans om bij een klasse te horen kan representeren.

Tijdens het trainen van het model zal er door middel van “binary_crossentropy” een ‘loss’ worden berekent op basis van hoe ver de percentuele waarde van de laatste sigmoid activatie af zit van het label (1.0 of 0.0) die hoort bij de input die het model gekregen heeft. Deze loss zal in combinatie met de optimizer ‘Adam’ de weights en discriminatoren van de neurons aanpassen door middel van ‘Back propogation’. Back propogation houdt simpel gezegd in dat de weights en discriminatore op terugwerkende kracht vanuit de laatste laag worden aangepast in het netwerk om te compenseren voor de loss.

Nu dat het ingewikkelde deel van het neurale netwerk is uitgelegd zal ik kort ingaan op de andere lagen van het netwerk. Er is een input laag met 64 neuronen met een relu activatie. Relu is de standaard keuze als activatie voor input en hidden layers en het representeert een lineaire functie waar alle waardes boven 0, dezelfde waarde blijven, maar alles onder 0 op 0 gezet wordt. De ‘input shape’ is de vorm van de data hoe het in het neurale netwerk komt. In mijn geval is de shape (8,) omdat de input bestaat uit 8 kolommen aan tabel data. De input shape zou een compleet andere vorm hebben voor andere typen data zoals bijvoorbeeld foto’s.

De hidden layer was in principe door de simpele opzet van de data niet echt nodig, maar een enkele hidden layer kan er wel voor zorgen dat het netwerk patronen herkent op basis van meerdere features in de input laag. Met deze reden heb ik een half zo kleine hidden layer toegevoegt. Over het algemeen heeft dit niet een zware impact gehad op hoe efficiënt het netwerk input naar output verwerkt en het model heeft geen last van ‘overfitting’ (het te erg trainen op training data waardoor het model slechter werkt op nieuwe data zoals de test/validatie set), dus er zitten geen negatieve consequenties aan het toevoegen van een hidden layer, ook al heeft het ook niet een duidelijke toegevoegde waarde in de evaluatie van het model.

Hiernaast is er een drop-out layer en dit zorgt er in basis alleen voor dat het model minder snel problemen krijgt met overfitting, want 50% van de activaties worden op 0 gezet. Deze hoeveelheid werkt zeer goed voor regulatie van het neurale netwerk.

Als laatste heb ik gekozen voor accuracy als metric, omdat de combinatie van accuracy en loss een goed beeld geeft over hoe goed je netwerk functioneert.

Trainen

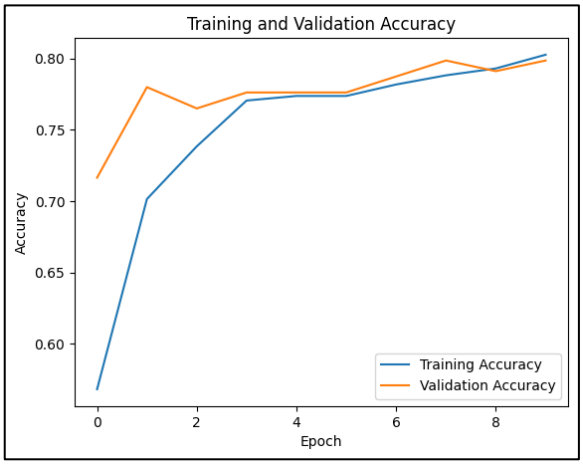
Zoals zichtbaar in de afbeelding hieronder heb ik mijn netwerk voor 10 epochs laten trainen.

```
# Train the model with validation data
history = model.fit(X_train, Y_train, epochs=10, validation_data=(X_test, Y_test), verbose=1)
✓ 1.3s

Epoch 1/10
20/20 [==============================] - 1s 9ms/step - loss: 0.6818 - accuracy: 0.5682 - val_loss: 0.6339 - val_accuracy: 0.7164
Epoch 2/10
20/20 [==============================] - 0s 3ms/step - loss: 0.6265 - accuracy: 0.7014 - val_loss: 0.5932 - val_accuracy: 0.7799
Epoch 3/10
20/20 [==============================] - 0s 3ms/step - loss: 0.5972 - accuracy: 0.7384 - val_loss: 0.5640 - val_accuracy: 0.7649
Epoch 4/10
20/20 [==============================] - 0s 3ms/step - loss: 0.5463 - accuracy: 0.7705 - val_loss: 0.5349 - val_accuracy: 0.7761
Epoch 5/10
20/20 [==============================] - 0s 3ms/step - loss: 0.5249 - accuracy: 0.7737 - val_loss: 0.5077 - val_accuracy: 0.7761
Epoch 6/10
20/20 [==============================] - 0s 3ms/step - loss: 0.5064 - accuracy: 0.7737 - val_loss: 0.4910 - val_accuracy: 0.7761
Epoch 7/10
20/20 [==============================] - 0s 3ms/step - loss: 0.4907 - accuracy: 0.7817 - val_loss: 0.4696 - val_accuracy: 0.7873
Epoch 8/10
20/20 [==============================] - 0s 3ms/step - loss: 0.4725 - accuracy: 0.7881 - val_loss: 0.4606 - val_accuracy: 0.7985
Epoch 9/10
20/20 [==============================] - 0s 3ms/step - loss: 0.4635 - accuracy: 0.7929 - val_loss: 0.4510 - val_accuracy: 0.7910
Epoch 10/10
20/20 [==============================] - 0s 4ms/step - loss: 0.4552 - accuracy: 0.8026 - val_loss: 0.4453 - val_accuracy: 0.7985
```

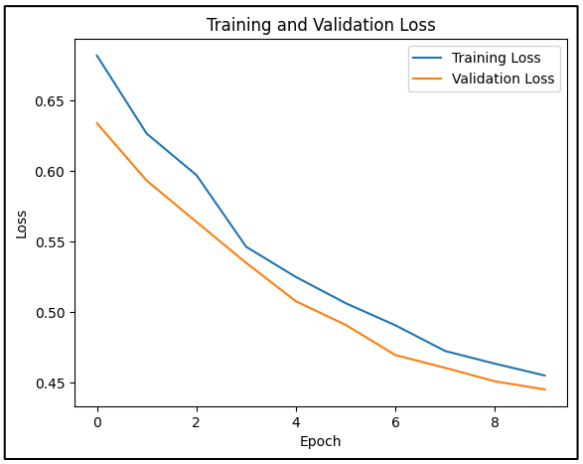
Epochs

Een enkele epoch houdt in dat je netwerk 1 keer traint op je complete dataset. Deze waarde heet een hyperparameter en dit houdt in dat deze waarde aangepast moet worden door de ontwikkelaar op basis van de evaluatie van het model. In mijn geval kwam er uit het train/validatie plot dat mijn netwerk begon te overfitten na 10 epochs en dus heb ik de training op deze hoeveelheid gezet. De grafieken waar de hyperparameters bepaald kunnen worden is hiernaast te zien en deze twee grafieken weergeven de performance over epochs.



Batch Size

Om een klein beetje dieper in te gaan op de nummers die hierboven te zien zijn, is er een laad balk die van 0 tot 20 loopt. De waardes 0 tot 20 lopen niet overeen met de hoeveelheid data in het netwerk en dat komt omdat een neuraal netwerk niet back propogation toepast op elk data punt. In plaats daarvan werkt een netwerk op basis van ‘batches’ van een aantal inputs in het netwerk. De maat van een batch kan handmatig worden meegegeven, maar voor deze opdracht was ik tevreden met de batch size, omdat het netwerk niet snel overfit, de training niet lang duurt en er een goede hoeveelheid datapunten staan in de evaluatie grafieken. Op terugwerkende kracht kunnen we de batch size berekenen door middel van “batch size = aantal samples/aantal batches” en in dit geval is dat ~45 samples per batch.



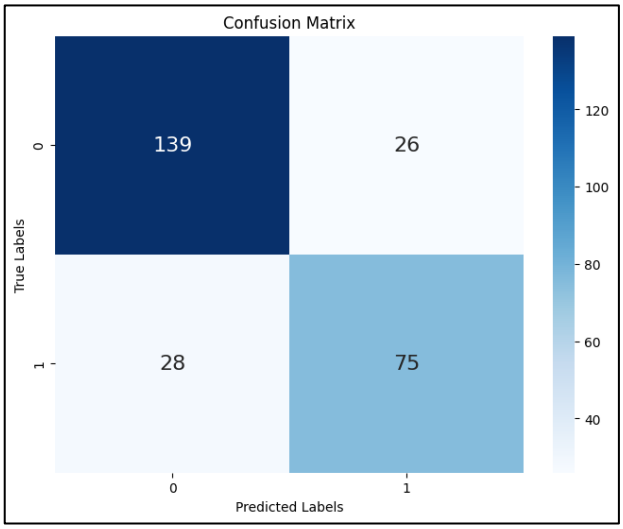
Validatie Set

Zoals ook zichtbaar in de training informatie en grafiek, is er gebruik gemaakt van een validatie set. Deze validatie set in mijn geval de testing set, omdat ik simpelweg een ruimer inzicht wilde hebben hoe mijn test set reageerde op het trainen van mijn model, maar bij een grotere kwantiteit van data is het gebruikelijk dat er eerst een 70/30 split wordt gedaan uit de data voor een train/test set en daarna kan er een validatie set gemaakt worden uit de overige 70% van de training set in nog een 70/30 verhouding, waardoor de training set 49% is, de validatie set 21% is en de testing set 30% is. Dit is echter zonde als de hoeveelheid data al relatief laag is. Een validatie set geeft inzichten in hoe nieuwe data reageert op verschillende epochs van het training proces en hiermee kun je overfitting diagnosticeren in je model.

Inzichten

Uit de training grafieken is voor mij duidelijk dat dit de optimale werking van het model is op basis van de beschikbare data van de dataset. Het model begint op een loss van ~0.65 en eindigt met een curve op een loss van ~0.45. Dit is een verbetering en laat zien dat het model inzichten heeft leren maken op basis van de data die beschikbaar was. Deze inzichten zijn op basis van de validatie kromme, aangezien dit representeert hoe goed het model reageert op data waar we nog niet de labels van zouden weten.

Resultaten



De resultaten van een neuraal netwerk kunnen weergeven worden op verschillende manieren voor verschillende soorten output lagen. Zo wordt er bij regressie vaak gebruik gemaakt van de mean-squared-error, terwijl er bij classificatie problemen vaak gebruik wordt gemaakt van een 'confusion matrix'. Dit is in principe niks anders als een matrix die de voorspelde labels tegen de echte labels van een test set aan zet. De confusion matrix die uit mijn oplossing kwam is dan ook hiernaast te zien. In mijn geval had heeft mijn model in de testing set 139 True Negatives (Sterven correct voorspeld), 75 True Positives (Overleven correct voorspeld), 26 False Positives (Overleven voorspeld terwijl de persoon in de werkelijkheid is overleden) en 28 False Negatives (Overlijden voorspeld terwijl de persoon in de werkelijkheid nog leefde).

Precision and Recall

Precision geeft aan hoeveel van de labels die zijn voorspeld voor een klasse ook echt correct waren en dit geeft een mate weer in hoe betrouwbaar de voorspellingen voor een klasse betrouwbaar zijn. In mijn geval was de precision van het model ~74,26%, wat aangeeft dat als mijn model voorspeld dat iemand zal overleven, dan is er een 74,26% kans dat deze voorspelling klopt. De precision voor een klasse is te bereken door middel van de berekening "TP/(TP + FP)".

Recall aan de andere kant geeft aan hoeveel van de klasse correct zijn opgepakt door de classificator. In mijn model was de recall ~72,82% en dit laat zien dat van de overlevenden, 72,82% van de mensen correct is ingedeeld onder de overlevende. De Recall voor een klasse is te berekenen door middel van de berekening "TP/(TP + FN)"

Deze twee waardes lijken in uitgesproken tekst vrij erg op elkaar, maar in de realiteit laat dit de twee manieren zien waarop de kracht van een classificator getest kan worden ten opzichte van de andere klassen van het systeem. Zo laat de precision zien hoe goed een classificator is in een bepaalde klassen op basis van de andere incorrect voorspelde waardes van de classificator, terwijl de recall laat zien hoe goed een classificator is in het opvangen van de klasse binnen de werkelijke labels. Dit zijn daarom ook de standaard evaluatie scores om een model op te keuren en als deze gecombineerd worden geeft dat een goed beeld in hoe effectief een model is in het voorspellen van een klasse.

F1-Score

Deze combinatie wordt weergeven met een F1-Score en deze score is in mijn model ~73,53% voor het voorspellen of iemand gaat overleven. Dit neemt beide de recall en precision in acht en is daarom een heel effectieve manier om een classificator te keuren. De F1-Score is te berekenen door middel van de berekening "(2 * Precision * Recall)/(Precision + Recall)"

Conclusie

Het model wat ik gemaakt heeft, heeft een F1-Score van ~73,53% voor het voorspellen of iemand zal overleven en een F1-Score van ~83,73% voor het voorspellen of iemand dood zal gaan. Dit verschil is niet meer als logisch voor een incident met zo'n grote hoeveelheid sterfgevallen waar ook een factor van chaos aanwezig was in de kwestie of iemand een grotere kans had om te overleven.

Een interessant punt is echter wel dat bijna 75% van de vrouwen aan boord van de titanic het hebben overleeft, terwijl 19% van de mannen het hebben overleeft. Mogelijk zou het interessant zijn om te kijken of een aparte classificator maken voor mannen en vrouwen een impact zou maken op de betrouwbaarheid van de twee classificatoren wanneer ze evenredig gebruikt worden op hun eigen klasse.

Daarentegen is de accuracy van het model over het algemeen ~80%, dus in dat opzicht is dit een betere classificator als gokken dat iedereen dood zou gaan (wat een accuracy van ~62,62% op zou leveren). Daarentegen, al zou men ervan uitgaan dat mannen doodgaan en vrouwen overleven, dan zou je al uitkomen op een accuracy van ~78,9%, waar mijn classificator maar een met een fractie bovenuit komt, maar dat is dan natuurlijk ook een patroon wat mijn neurale netwerk herkent en meeneemt in zijn oordeel.

Dit patroon zou naar boven komen al zou men een 'Random Tree' classificatie gebruiken in plaats van een Neuraal Netwerk en daarom zou ik in andere gevallen ook sneller bij tabel data kiezen voor een Random Tree/Forest classificatie.