

Review

Neural Networks for Machine Learning Applications

Spring 2023

Sakari Lukkarinen

Metropolia University of Applied Sciences

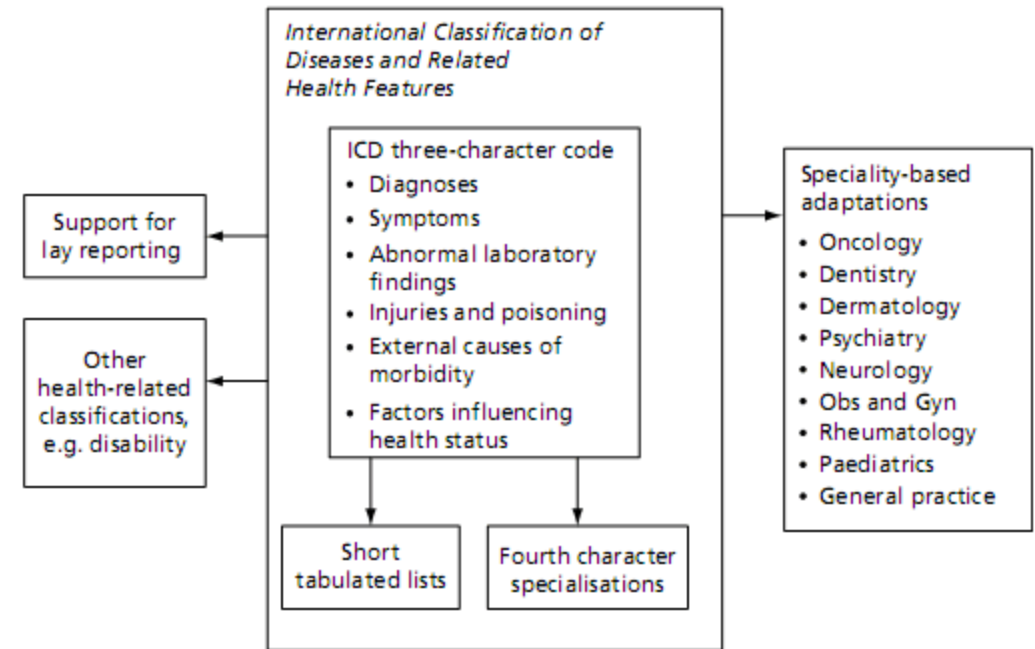


Topics

- Classification
 - Basic Dense Neural Network (DNN)
 - Case 1
- Image processing
 - Convolutional Neural Networks (CNN)
 - Case 2
- Text processing
 - RNN, LSTM, word embeddings
 - Case 3
- Model evaluation
- Datasets and metrics

Classification problem

- Does this patient have
 - cardiovascular disease? (Case 1)
 - pneumonia? (Case 2)
- Is this drug review
 - Negative, neutral or positive? (Case 3)
- More generic
 - Does this patient need medical treatment?
 - What would be the best medical treatment for this patient?



[Overview of International Classification of Diseases \(ICD\)](#)

Basic classifier (Dense Neural Network)

```
from tensorflow.keras import Sequential, layers

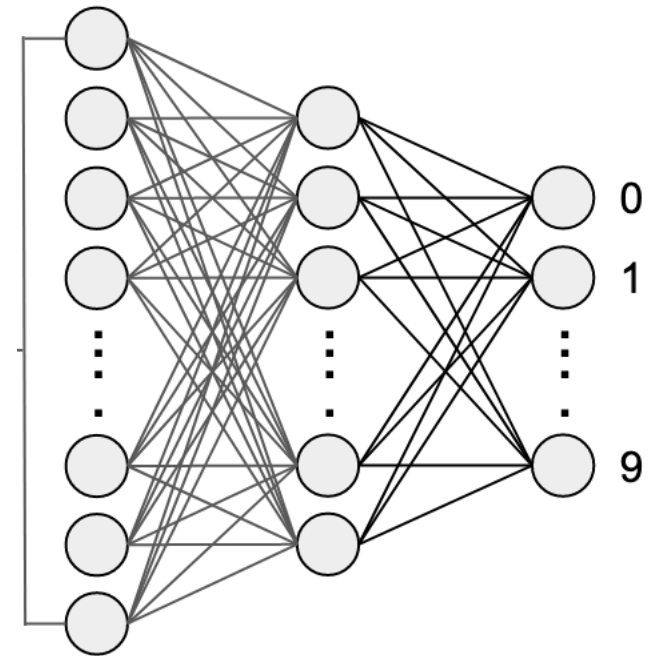
model = Sequential([
    layers.Input(shape = (13, )),
    layers.Dense(4, activation="relu"),
    layers.Dense(1, activation="sigmoid")
])

model.compile(optimizer="rmsprop", loss="binary_crossentropy", metrics=["accuracy"])

model.summary()
```

Model: "sequential"

Layer (type)	Output Shape	Param #
=====		
dense (Dense)	(None, 4)	56
=====		
dense_1 (Dense)	(None, 1)	5
=====		
Total params: 61		
Trainable params: 61		
Non-trainable params: 0		
=====		



Optimizers

This is how the model is updated based on the data it sees and its loss function.

Classes

`class Adadelta`: Optimizer that implements the Adadelta algorithm.

`class Adagrad`: Optimizer that implements the Adagrad algorithm.

`class Adam`: Optimizer that implements the Adam algorithm.

`class Adamax`: Optimizer that implements the Adamax algorithm.

`class Ftrl`: Optimizer that implements the FTRL algorithm.

`class Nadam`: Optimizer that implements the NAdam algorithm.

`class Optimizer`: Base class for Keras optimizers.

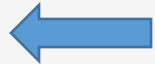
`class RMSprop`: Optimizer that implements the RMSprop algorithm.

`class SGD`: Gradient descent (with momentum) optimizer.

```
from tensorflow.keras.optimizers import RMSprop, Adam
```

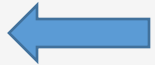
```
# Use alias
```

```
model.compile(optimizer = "rmsprop",  
              loss = "binary_crossentropy",  
              metrics = ["accuracy"])
```



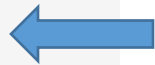
```
# Use class
```

```
model.compile(optimizer = RMSprop(),  
              loss = "binary_crossentropy",  
              metrics = ["accuracy"])
```



```
# Use class with arguments
```

```
model.compile(optimizer = Adam(learning_rate = 0.0001),  
              loss = "binary_crossentropy",  
              metrics = ["accuracy"])
```



Losses

The purpose of loss functions is to compute the quantity that a model should seek to minimize during training.

Classes

`class BinaryCrossentropy` : Computes the cross-entropy loss between true labels and predicted labels.

`class CategoricalCrossentropy` : Computes the crossentropy loss between the labels and predictions.

`class CategoricalHinge` : Computes the categorical hinge loss between `y_true` and `y_pred`.

`class CosineSimilarity` : Computes the cosine similarity between labels and predictions.

`class Hinge` : Computes the hinge loss between `y_true` and `y_pred`.

`class Huber` : Computes the Huber loss between `y_true` and `y_pred`.

`class KLDivergence` : Computes Kullback-Leibler divergence loss between `y_true` and `y_pred`.

`class LogCosh` : Computes the logarithm of the hyperbolic cosine of the prediction error.

`class Loss` : Loss base class.

`class MeanAbsoluteError` : Computes the mean of absolute difference between labels and predictions.

```
from tensorflow.keras.losses import BinaryCrossentropy

# Use aliases
model.compile(optimizer = "rmsprop",
              loss = "binary_crossentropy", ←
              metrics = ["accuracy"])

# Use classes
model.compile(optimizer = "rmsprop",
              loss = BinaryCrossentropy(), ←
              metrics = ["accuracy"])
```

https://www.tensorflow.org/api_docs/python/tf/keras/losses

Metrics

A metric is a function that is used to judge the performance of your model.

Classes

`class AUC` : Computes the approximate AUC (Area under the curve) via a Riemann sum.

`class Accuracy` : Calculates how often predictions equal labels.

`class BinaryAccuracy` : Calculates how often predictions match binary labels.

`class BinaryCrossentropy` : Computes the crossentropy metric between the labels and predictions.

`class CategoricalAccuracy` : Calculates how often predictions matches one-hot labels.

`class CategoricalCrossentropy` : Computes the crossentropy metric between the labels and predictions.

`class CategoricalHinge` : Computes the categorical hinge metric between `y_true` and `y_pred`.

`class CosineSimilarity` : Computes the cosine similarity between the labels and predictions.

`class FalseNegatives` : Calculates the number of false negatives.

`class FalsePositives` : Calculates the number of false positives.

BinaryAccuracy and CategoricalAccuracy are the most commonly used metrics for binary (2 classes) or categorical (N classes) classification problems.

Alias for both of these is `metrics = ["accuracy"]`

https://www.tensorflow.org/api_docs/python/tf/keras/metrics

Convolutional Neural Networks (CNN)

Convolutional Neural Network (CNN)

A [neural network](#) in which at least one layer is a [convolutional layer](#). A typical convolutional neural network consists of some combination of the following layers:

- [convolutional layers](#)
- [pooling layers](#)
- [dense layers](#)

Convolutional neural networks have had great success in certain kinds of problems, such as image recognition

```
model = models.Sequential()  
model.add(layers.Conv2D(32, (3, 3), activation='relu', input_shape=(32, 32, 3)))  
model.add(layers.MaxPooling2D((2, 2)))  
model.add(layers.Conv2D(64, (3, 3), activation='relu'))  
model.add(layers.MaxPooling2D((2, 2)))  
model.add(layers.Conv2D(64, (3, 3), activation='relu'))
```

Let's display the architecture of our model so far.

```
model.summary()
```

Model: "sequential"

Layer (type)	Output Shape	Param #

conv2d (Conv2D)	(None, 30, 30, 32)	896

max_pooling2d (MaxPooling2D)	(None, 15, 15, 32)	0

conv2d_1 (Conv2D)	(None, 13, 13, 64)	18496

max_pooling2d_1 (MaxPooling2D)	(None, 6, 6, 64)	0

conv2d_2 (Conv2D)	(None, 4, 4, 64)	36928

Image processing with Neural Networks

- Small images can be flattened as 1D arrays
- Large images typically need *data generators* and *batching*
- Usually all images are *rescaled*
- If there are few images, you can use *data augmentation*

```
train_datagen = ImageDataGenerator(rescale=1./255,  
                                   validation_split = 0.2,  
                                   rotation_range = 5)  
  
train_generator = train_datagen.flow_from_directory(  
    train_dir,  
    target_size = (150, 150),  
    batch_size=16,  
    class_mode='binary',  
    subset = "training")  
  
dev_generator = train_datagen.flow_from_directory(  
    train_dir,  
    target_size = (150, 150),  
    batch_size=16,  
    class_mode='binary',  
    subset = "validation")
```

```
Found 4187 images belonging to 2 classes.  
Found 1045 images belonging to 2 classes.
```

Data augmentation example

```
data_augmentation = keras.Sequential(  
    [  
        layers.experimental.preprocessing.RandomFlip("horizontal",  
                                                    input_shape=(img_height,  
                                                                img_width,  
                                                                3)),  
        layers.experimental.preprocessing.RandomRotation(0.1),  
        layers.experimental.preprocessing.RandomZoom(0.1),  
    ]  
)
```

https://www.tensorflow.org/tutorials/images/classification#data_augmentation

Advanced image preprocessing (extra)

```
def process_data_train(image_path, label):  
  
    img = tf.io.read_file(image_path)  
    img = tf.image.decode_jpeg(img, channels = 0)  
    img = tf.image.resize(img, IMG_SIZE)  
    # img = tf.image.random_brightness(img, 0.3)  
    # img = tf.image.random_flip_left_right(img)  
    # img = tf.image.per_image_standardization(img)  
  
    return img, label  
  
def configure_training(ds, batch_size = 64):  
  
    ds = ds.cache('/kaggle/dump.train')  
    ds = ds.repeat()  
    ds = ds.shuffle(buffer_size = 1024)  
    ds = ds.batch(batch_size)  
    ds = ds.prefetch(buffer_size = tf.data.experimental.AUTOTUNE)  
  
    return ds
```

```
pos = df[labels[n]] == 1  
paths = df['path'][pos].values  
outputs = df[labels][pos]  
pos_ds = tf.data.Dataset.from_tensor_slices((paths, outputs))  
pos_ds = pos_ds.map(process_data_train, num_parallel_calls = tf.data.experimental.AUTOTUNE)
```

`Dataset.from_tensor_slices(...)` creates a dataset generator.

- Path = filenames for input images
- Output = labels

`.map()` calls to `process_data_train()` to process the images and create a flow of images with labels.

`Configure_training()` optimizes the flow for GPU processing.

Text processing

Text processing

The text needs to be converted to numbers. Most common ways to do that are *one-hot encoding*, *word embeddings* and *word2vec*

One-hot encoding

	cat	mat	on	sat	the
the =>	0	0	0	0	1
cat =>	1	0	0	0	0
sat =>	0	0	0	1	0
...					

A 4-dimensional embedding

cat =>	1.2	-0.1	4.3	3.2
mat =>	0.4	2.5	-0.9	0.5
on =>	2.1	0.3	0.1	0.4
...				

https://www.tensorflow.org/tutorials/text/word_embeddings

<https://www.tensorflow.org/tutorials/text/word2vec>

Example – Case 3

- One-hot coding

```
# Tokenize the text
samples = train['review']
tokenizer = Tokenizer(num_words = 5000)
tokenizer.fit_on_texts(samples)

# Make one hot samples
data = tokenizer.texts_to_matrix(samples, mode='binary')
```

```
# What is the size of the dataset?
data.shape
```

```
(15000, 5000)
```

Word embedding

```
%%time
# Tokenize the text
samples = train['review']
tokenizer = Tokenizer(num_words = 5000)
tokenizer.fit_on_texts(samples)
# Convert text to sequences
sequences = tokenizer.texts_to_sequences(samples)

word_index = tokenizer.word_index
print('Found {} unique tokens.'.format(len(word_index)))
```

```
Found 19946 unique tokens.
CPU times: user 2.43 s, sys: 9.74 ms, total: 2.44 s
Wall time: 2.44 s
```

```
data = pad_sequences(sequences, maxlen=200)
```

Evaluation of performance

Underfit and overfit

The aim: to develop models that generalize well to a *testing set* (or data they haven't seen before)

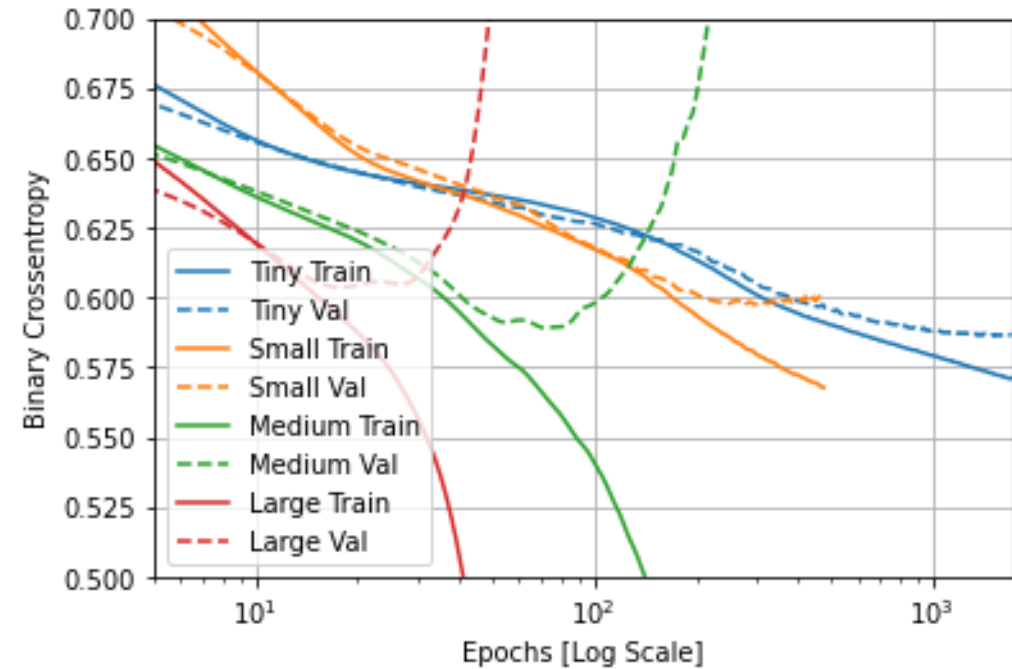
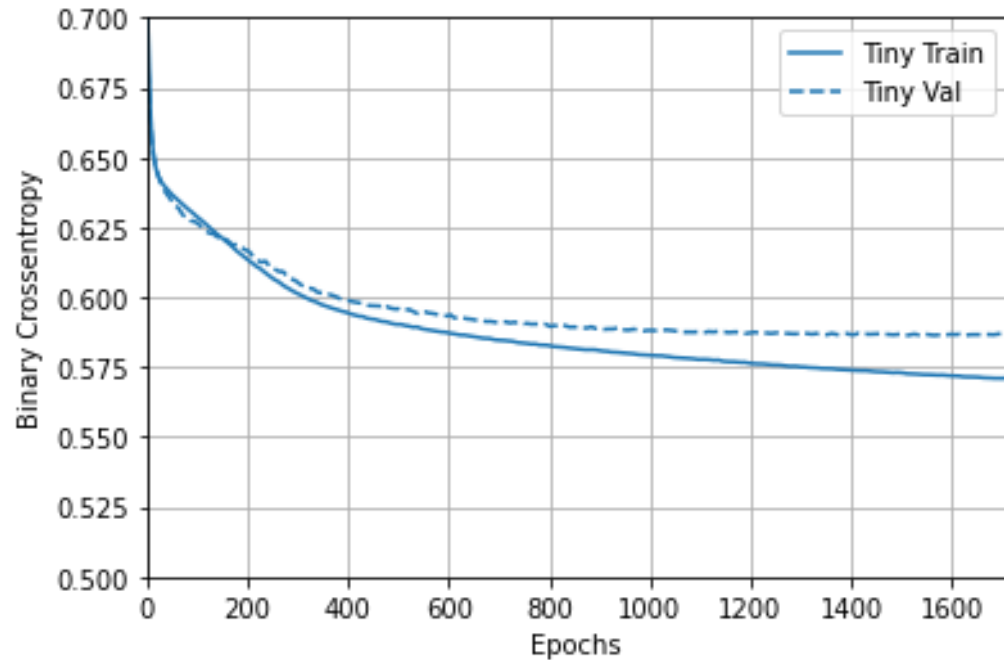
Underfitting occurs when there is still room for improvement on the train data. The network has not learned the relevant patterns in the training data. This can happen for a number of reasons:

- model is not powerful enough
- is over-regularized
- has simply not been trained long enough

If you train for too long though, the model will start to *overfit* and learn patterns from the training data that don't generalize to the test data. To prevent overfitting:

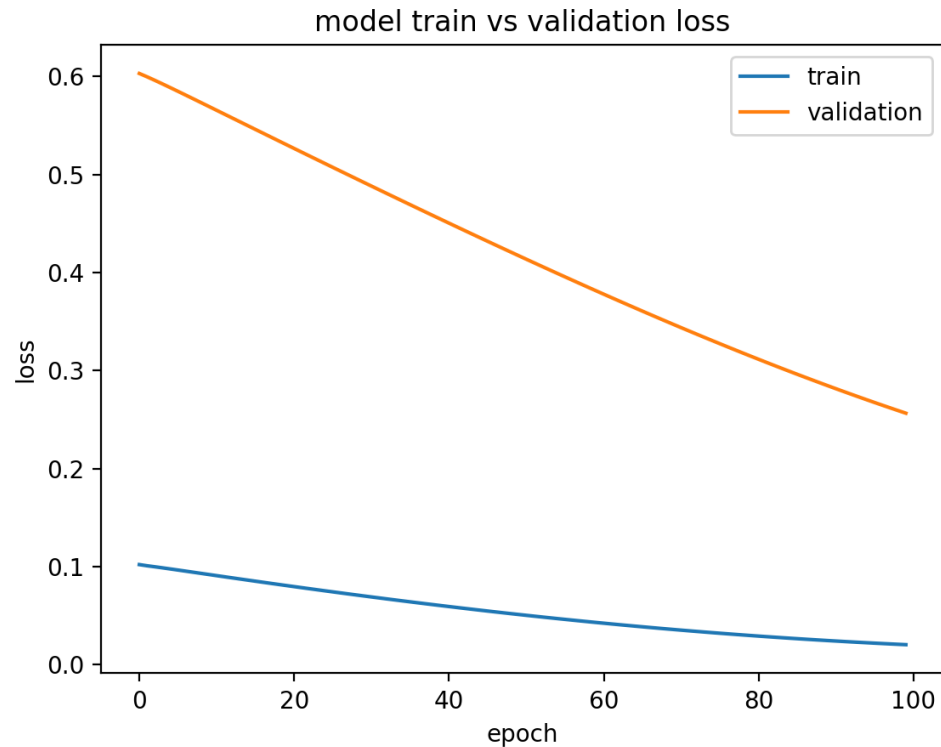
- The best solution is to use more complete training data
 - The dataset should cover the full range of inputs that the model is expected to handle. Additional data may only be useful if it covers new and interesting cases.
- The next best solution is to use techniques, like regularization
 - These place constraints on the quantity and type of information your model can store. If a network can only afford to memorize a small number of patterns, the optimization process will force it to focus on the most prominent patterns, which have a better chance of generalizing well.

Examples of underfit and overfit

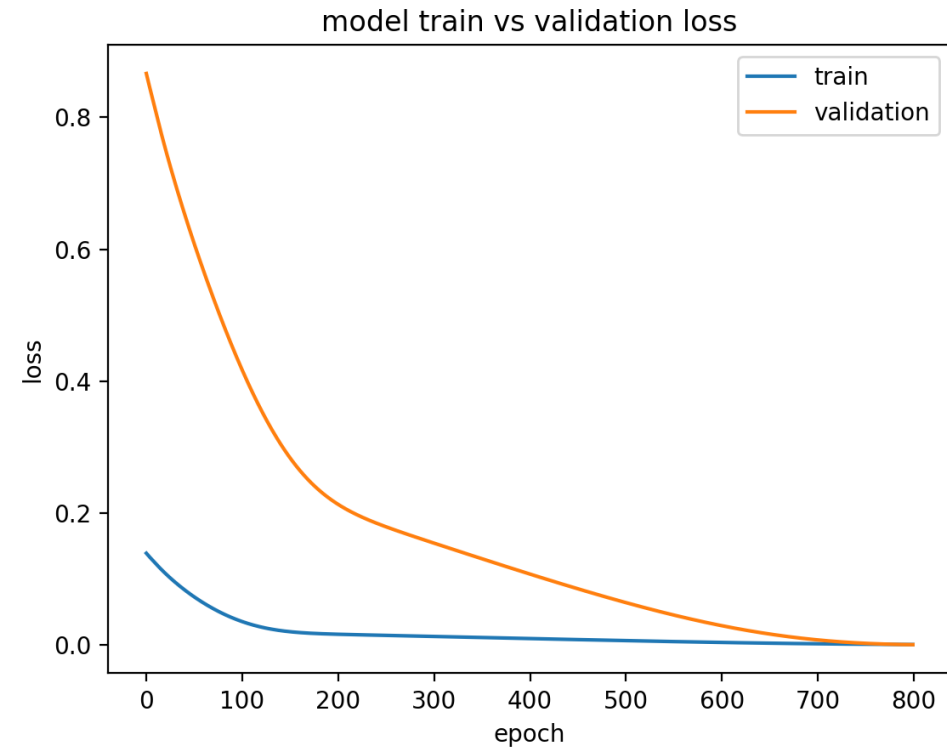


https://www.tensorflow.org/tutorials/keras/overfit_and_underfit

Underfitting vs. good fitting

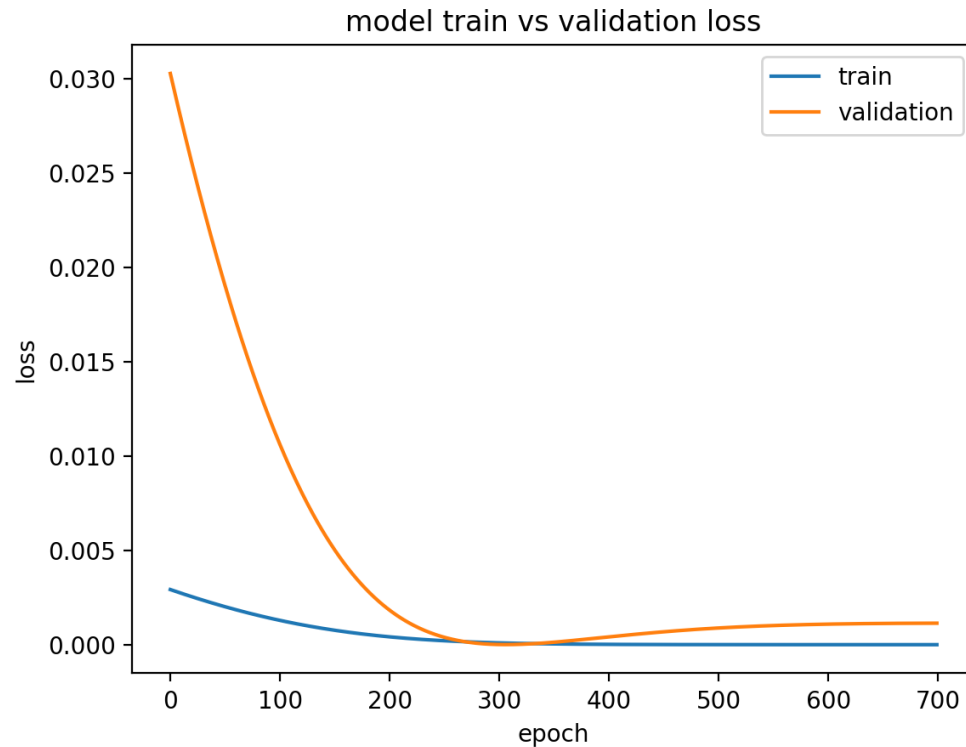


Underfitting

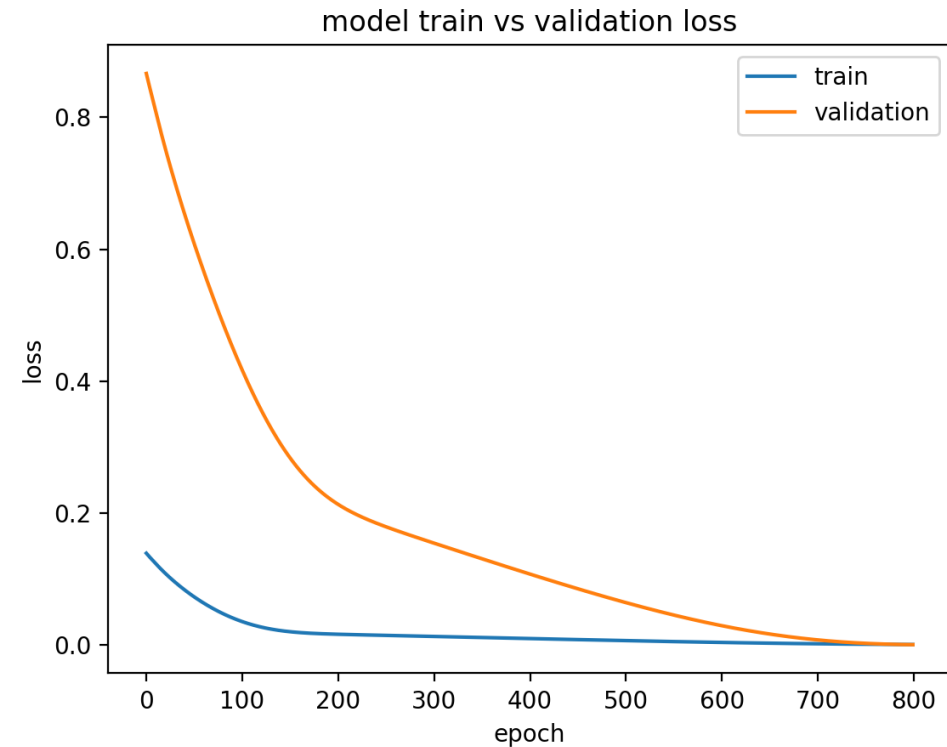


Good model

Overfitting vs. good fit



Overfitting



Good model

<https://machinelearningmastery.com/diagnose-overfitting-underfitting-lstm-models/>

Datasets and Metrics

Datasets and Metrics

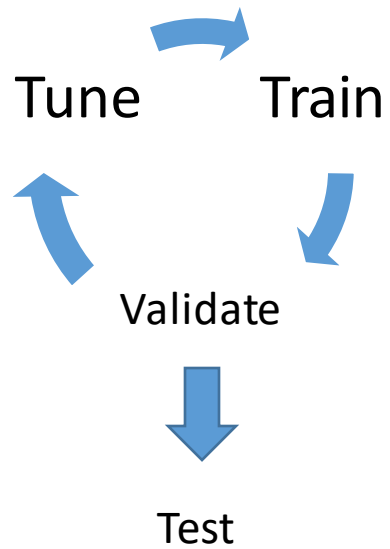
- Dataset

- Training, validation and test set
- Supervised learning process
- Why we need separate validation and test sets
- How should we split the original data into datasets
- 80/20 rule

- Metrics

- Accuracy
- Medical tests
 - Sensitivity and specificity
- Precision and recall
- Confusion matrix
- ROC curve
- Performance metrics in scikit-learn
 - Confusion matrix
 - Precision, recall, fscore, and support
 - Classification report

Setting up development and test sets



We usually define:

- **Training set** — Which you run your learning algorithm on.
- **Dev (development) set** — Which you use to tune parameters, select features, and make other decisions regarding the learning algorithm. Sometimes also called the **hold-out cross validation set**.
- **Test set** — which you use to evaluate the performance of the algorithm, but not to make any decisions regarding what learning algorithm or parameters to use.

Once you define a dev set (development set) and test set, your team will try a lot of ideas, such as different learning algorithm parameters, to see what works best. The dev and test sets allow your team to quickly see how well your algorithm is doing.

In other words, **the purpose of the dev and test sets are to direct your team toward the most important changes to make to the machine learning system.**

Most common metrics

Classification metrics

- **Accuracy**, precision, recall, fscore
- **Sensitivity and specificity**
- ROC Curve and area under ROC

Methods for classification prediction results

- **Confusion Matrix**
- **Classification Report**

Regression metrics

- Mean Absolute Error (MAE)
- Mean Squared Error (MSE)
- R^2 (R-squared)

Confusion matrix [\[edit \]](#)

Let us consider a group with **P** positive instances and **N** negative instances of some condition. The four outcomes can be formulated in a 2×2 *contingency table* or *confusion matrix*, as follows:

		True condition			
Total population		Condition positive	Condition negative	Prevalence = $\frac{\Sigma \text{ Condition positive}}{\Sigma \text{ Total population}}$	Accuracy (ACC) = $\frac{\Sigma \text{ True positive} + \Sigma \text{ True negative}}{\Sigma \text{ Total population}}$
Predicted condition	Predicted condition positive	True positive, Power	False positive, Type I error	Positive predictive value (PPV), Precision = $\frac{\Sigma \text{ True positive}}{\Sigma \text{ Predicted condition positive}}$	False discovery rate (FDR) = $\frac{\Sigma \text{ False positive}}{\Sigma \text{ Predicted condition positive}}$
	Predicted condition negative	False negative, Type II error	True negative	False omission rate (FOR) = $\frac{\Sigma \text{ False negative}}{\Sigma \text{ Predicted condition negative}}$	Negative predictive value (NPV) = $\frac{\Sigma \text{ True negative}}{\Sigma \text{ Predicted condition negative}}$
		True positive rate (TPR), Recall, Sensitivity, probability of detection = $\frac{\Sigma \text{ True positive}}{\Sigma \text{ Condition positive}}$	False positive rate (FPR), Fall-out, probability of false alarm = $\frac{\Sigma \text{ False positive}}{\Sigma \text{ Condition negative}}$	Positive likelihood ratio (LR+) = $\frac{\text{TPR}}{\text{FPR}}$	Diagnostic odds ratio (DOR) = $\frac{\text{LR+}}{\text{LR-}}$
		False negative rate (FNR), Miss rate = $\frac{\Sigma \text{ False negative}}{\Sigma \text{ Condition positive}}$	True negative rate (TNR), Specificity (SPC) = $\frac{\Sigma \text{ True negative}}{\Sigma \text{ Condition negative}}$	Negative likelihood ratio (LR-) = $\frac{\text{FNR}}{\text{TNR}}$	
				F ₁ score = $\frac{2}{\frac{1}{\text{Recall}} + \frac{1}{\text{Precision}}}$	

3.3.2. Classification metrics

The `sklearn.metrics` module implements several loss, score, and utility functions to measure classification performance. Some metrics might require probability estimates of the positive class, confidence values, or binary decisions values. Most implementations allow each sample to provide a weighted contribution to the overall score, through the `sample_weight` parameter.

Some of these are restricted to the binary classification case:

<code>precision_recall_curve</code> (<code>y_true</code> , <code>probas_pred</code>)	Compute precision-recall pairs for different probability thresholds
<code>roc_curve</code> (<code>y_true</code> , <code>y_score</code> [, <code>pos_label</code> , ...])	Compute Receiver operating characteristic (ROC)

Others also work in the multiclass case:

<code>cohen_kappa_score</code> (<code>y1</code> , <code>y2</code> [, <code>labels</code> , <code>weights</code> , ...])	Cohen's kappa: a statistic that measures inter-annotator agreement.
<code>confusion_matrix</code> (<code>y_true</code> , <code>y_pred</code> [, <code>labels</code> , ...])	Compute confusion matrix to evaluate the accuracy of a classification
<code>hinge_loss</code> (<code>y_true</code> , <code>pred_decision</code> [, <code>labels</code> , ...])	Average hinge loss (non-regularized)
<code>matthews_corrcoef</code> (<code>y_true</code> , <code>y_pred</code> [, ...])	Compute the Matthews correlation coefficient (MCC)

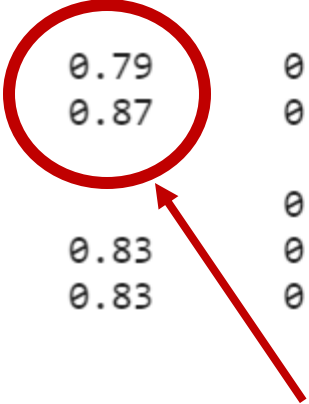
http://scikit-learn.org/stable/modules/model_evaluation.html#classification-metrics

Classification report

[Classification report](#) builds a text report showing the main classification metrics

```
CR = classification_report(test_labels, pred_labels)
print(CR)
```

	precision	recall	f1-score	support
0	0.84	0.79	0.82	87
1	0.82	0.87	0.84	97
accuracy			0.83	184
macro avg	0.83	0.83	0.83	184
weighted avg	0.83	0.83	0.83	184



For binary classification problem these are the sensitivity and specificity metrics values!