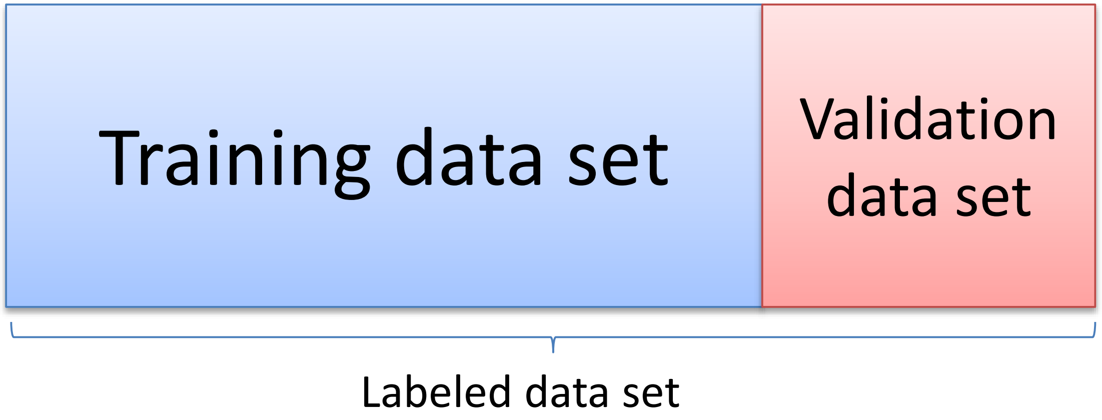
- **Validation methods**
  - **Holdout validation**
  - **K-fold cross validation**
- **Data preprocessing**
- **Feature engineering**
- **Overfitting and underfitting**
  - **Kernel regularization**
  - **Dropout**

Evaluating Machine learning models: Hold-out Validation

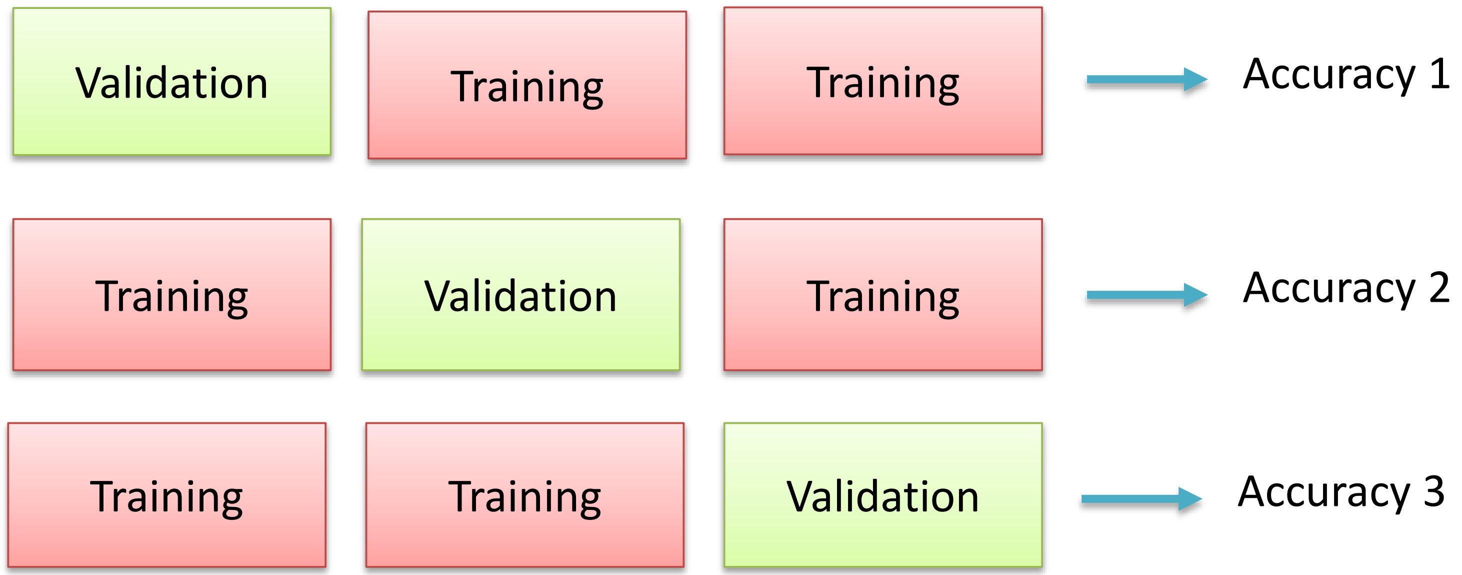
- In order to train and evaluate a model, the available data are often divided into training, validation and test sets.
- While the model is trained with training data, it is also evaluated with verification data.
- In the example below, training data is divided into training and validation. Thus, the performance of the model is measured with the epoch validation set during the training.



```
validation_size= 10000
#Shuffle dataset
np.random.shuffle(veriseti)
#Use some of the dataset for validation
validation_dataset = dataset[  : validation_size]
#Use the remaining for validation
training_ dataset = dataset[validation_size:  ]
```

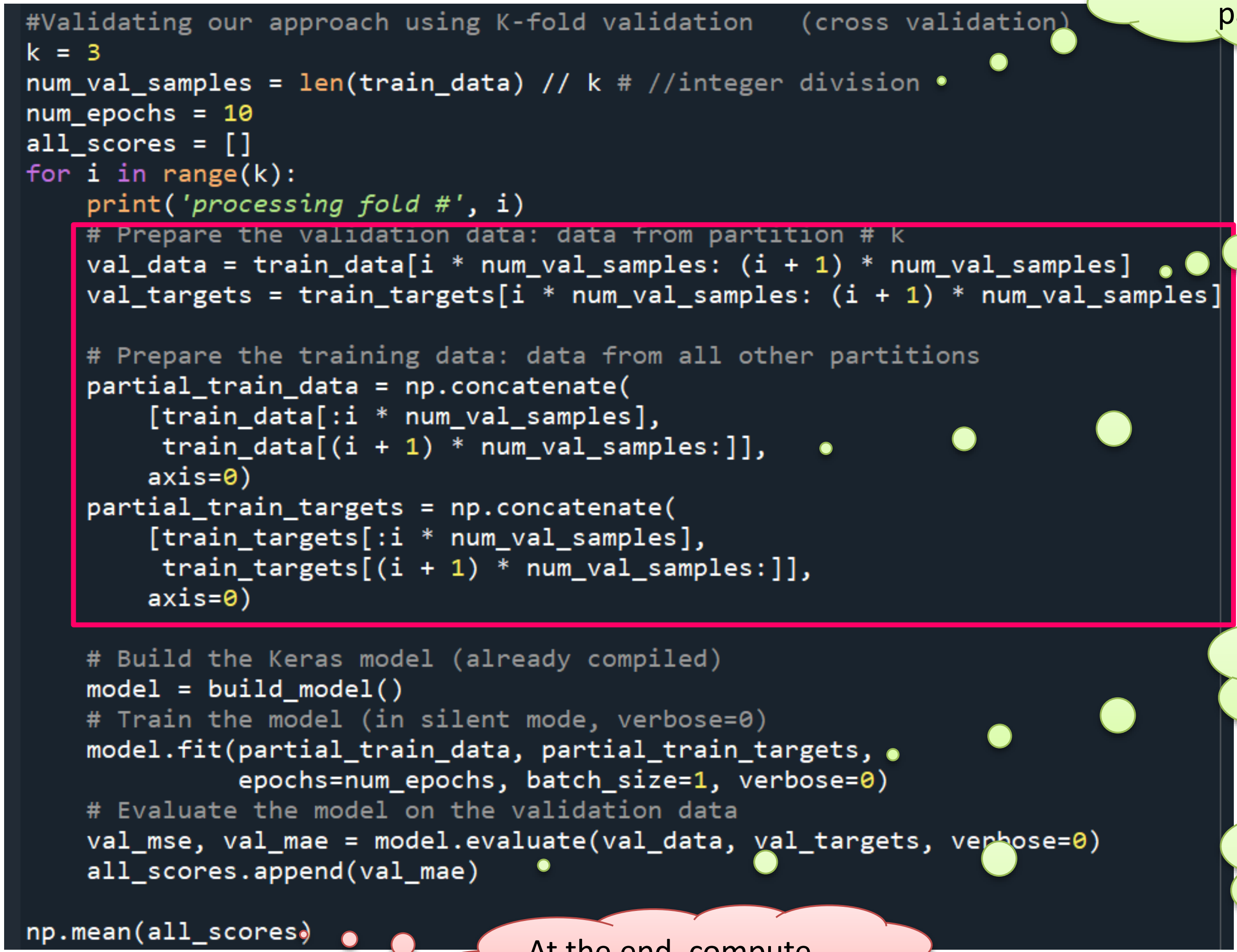
K-Fold Validation

- In this approach, the data is divided into K equal parts.
- One of the parts is used in verification and the others in training.
- This process is repeated for each part and the accuracy is calculated.
- For example, training data is divided into three parts and each part is used for validation.



Average Accuracy= (Accuracy 1+ Accuracy 2+ Accuracy 3)/3

Example: Boston house price prediction



```python
#Validating our approach using K-fold validation   (cross validation)
k = 3
num_val_samples = len(train_data) // k # //integer division
num_epochs = 10
all_scores = []
for i in range(k):
    print('processing fold #', i)
    # Prepare the validation data: data from partition # k
    val_data = train_data[i * num_val_samples: (i + 1) * num_val_samples]
    val_targets = train_targets[i * num_val_samples: (i + 1) * num_val_samples]

    # Prepare the training data: data from all other partitions
    partial_train_data = np.concatenate(
        [train_data[:i * num_val_samples],
         train_data[(i + 1) * num_val_samples:]],
        axis=0)
    partial_train_targets = np.concatenate(
        [train_targets[:i * num_val_samples],
         train_targets[(i + 1) * num_val_samples:]],
        axis=0)

    # Build the Keras model (already compiled)
    model = build_model()
    # Train the model (in silent mode, verbose=0)
    model.fit(partial_train_data, partial_train_targets,
              epochs=num_epochs, batch_size=1, verbose=0)
    # Evaluate the model on the validation data
    val_mse, val_mae = model.evaluate(val_data, val_targets, verbose=0)
    all_scores.append(val_mae)

np.mean(all_scores)
```

Divide the dataset into k parts

Select one part of data for validation

Use remaining data for training

Train model for the selected training dataset

Evaluate validation set and store accuracy

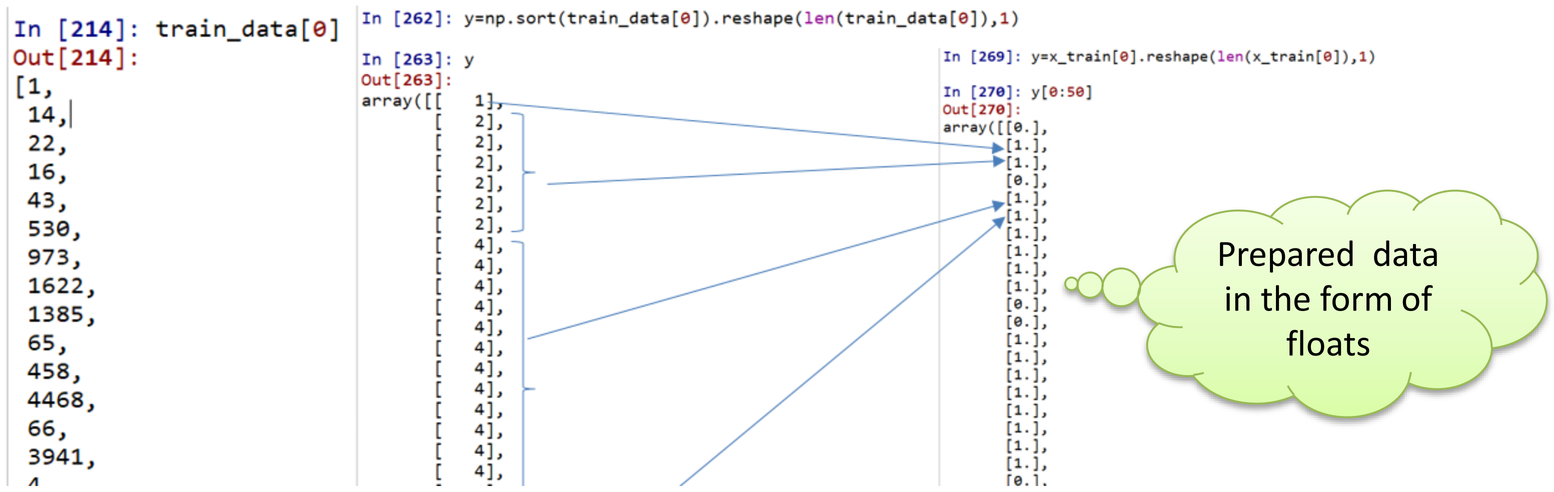
At the end, compute average score

Data preprocessing

- Common types:
  - Conversion to tensor representation
  - Normalization
  - Completing the missing values

Data preprocessing: Tensor representation

- Inputs and outputs must be represented with float or integer tensors.
- Data needed to process such as sound, images, text should be first turned into tensors, a step called data vectorization.
- For instance, in text-classification examples, we started from text represented as lists of integers (standing for sequences of words), and we used one-hot encoding to turn them into a tensor of float32 data.
- In the examples of classifying digits and predicting house prices, the data already came in tensor form, so you were able to skip this step



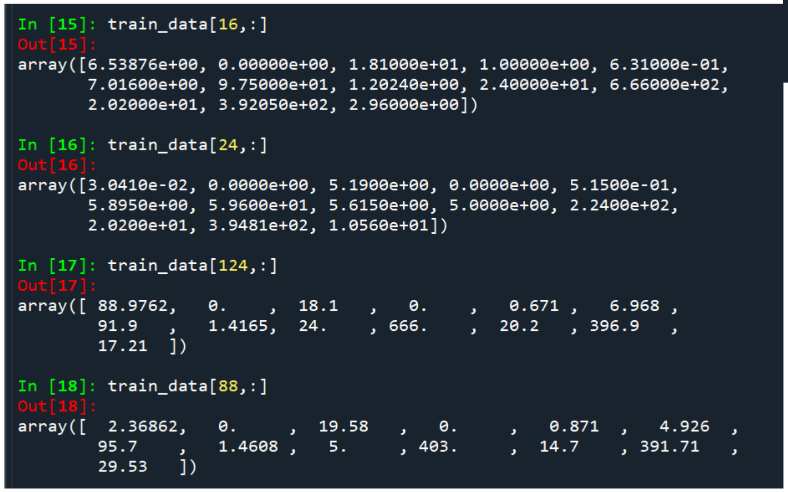
```
In [214]: train_data[0]
Out[214]:
[1,
 14,
 22,
 16,
 43,
 530,
 973,
 1622,
 1385,
 65,
 458,
 4468,
 66,
 3941,
 4
```

```
In [262]: y=np.sort(train_data[0]).reshape(len(train_data[0]),1)

In [263]: y
Out[263]:
array([[   1],
       [   2],
       [   2],
       [   2],
       [   2],
       [   2],
       [   2],
       [   4],
       [   4],
       [   4],
       [   4],
       [   4],
       [   4],
       [   4],
       [   4],
       [   4],
       [   4],
       [   4],
       [   4],
```

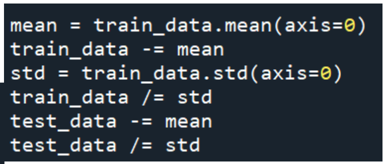
```
In [269]: y=x_train[0].reshape(len(x_train[0]),1)

In [270]: y[0:50]
Out[270]:
array([[0.],
       [1.],
       [1.],
       [0.],
       [1.],
       [1.],
       [1.],
       [1.],
       [1.],
       [1.],
       [0.],
       [0.],
       [1.],
       [1.],
       [1.],
       [1.],
       [1.],
       [1.],
       [1.],
       [1.],
       [0.],
```

Value Normalization

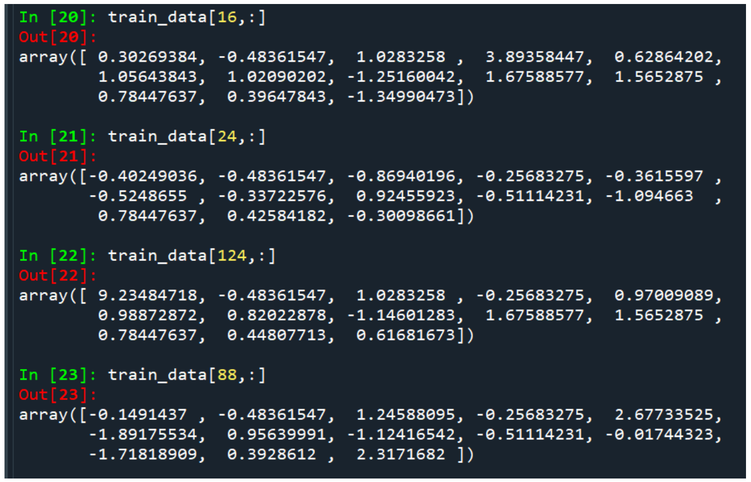
- In the digit-classification example, you started from image data encoded as integers in the 0–255 range, encoding grayscale values.
- Before you fed this data into your network, you had to cast it to float32 and divide by 255 so you'd end up with floating point values in the 0–1 range.
- Similarly, when predicting house prices, you started from features that took a variety of ranges—some features had small floating-point values, others had fairly large integer values.
- Before you fed this data into your network, you had to normalize each feature independently so that it had a standard deviation of 1 and a mean of 0



```
In [15]: train_data[16,:]
Out[15]:
array([6.53876e+00, 0.00000e+00, 1.81000e+01, 1.00000e+00, 6.31000e-01,
       7.01600e+00, 9.75000e+01, 1.20240e+00, 2.40000e+01, 6.66000e+02,
       2.02000e+01, 3.92050e+02, 2.96000e+00])

In [16]: train_data[24,:]
Out[16]:
array([3.0410e-02, 0.0000e+00, 5.1900e+00, 0.0000e+00, 5.1500e-01,
       5.8950e+00, 5.9600e+01, 5.6150e+00, 5.0000e+00, 2.2400e+02,
       2.0200e+01, 3.9481e+02, 1.0560e+01])

In [17]: train_data[124,:]
Out[17]:
array([ 88.9762,   0.    ,  18.1   ,   0.     ,   0.671 ,   6.968 ,
        91.9   ,   1.4165,  24.    , 666.     ,  20.2   , 396.9   ,
        17.21  ])

In [18]: train_data[88,:]
Out[18]:
array([  2.36862,   0.     ,  19.58   ,   0.     ,   0.871  ,   4.926  ,
        95.7    ,   1.4608 ,   5.     , 403.     ,  14.7    , 391.71   ,
        29.53    ])
```



```
mean = train_data.mean(axis=0)
train_data -= mean
std = train_data.std(axis=0)
train_data /= std
test_data -= mean
test_data /= std
```



```
In [20]: train_data[16,:]
Out[20]:
array([ 0.30269384, -0.48361547,  1.0283258 ,  3.89358447,  0.62864202,
        1.05643843,  1.02090202, -1.25160042,  1.67588577,  1.5652875 ,
        0.78447637,  0.39647843, -1.34990473])

In [21]: train_data[24,:]
Out[21]:
array([-0.40249036, -0.48361547, -0.86940196, -0.25683275, -0.3615597 ,
       -0.5248655 , -0.33722576,  0.92455923, -0.51114231, -1.094663  ,
        0.78447637,  0.42584182, -0.30098661])

In [22]: train_data[124,:]
Out[22]:
array([ 9.23484718, -0.48361547,  1.0283258 , -0.25683275,  0.97009089,
        0.98872872,  0.82022878, -1.14601283,  1.67588577,  1.5652875 ,
        0.78447637,  0.44807713,  0.61681673])

In [23]: train_data[88,:]
Out[23]:
array([-0.1491437 , -0.48361547,  1.24588095, -0.25683275,  2.67733525,
       -1.89175534,  0.95639991, -1.12416542, -0.51114231, -0.01744323,
       -1.71818909,  0.3928612 ,  2.3171682 ])
```

Value Normalization

- In general, it isn’t safe to feed into a neural network data that takes relatively large values (for example, multidigit integers, which are much larger than the initial values taken by the weights of a network) or data that is heterogeneous (for example, data where one feature is in the range 0–1 and another is in the range 100–200). Doing so can trigger large gradient updates that will prevent the network from converging.
- To make learning easier for your network, your data should have the following characteristics:
  - *Take small values*—Typically, most values should be in the 0–1 range.
- *Be homogenous*—That is, all features should take values in roughly the same range.
- Additionally, the following stricter normalization practice is common and can help, although it isn’t always necessary (for example, you didn’t do this in the digit-classification example):
  - Normalize each feature independently to have a mean of 0.
  - Normalize each feature independently to have a standard deviation of 1.
- This is easy to do with Numpy arrays:

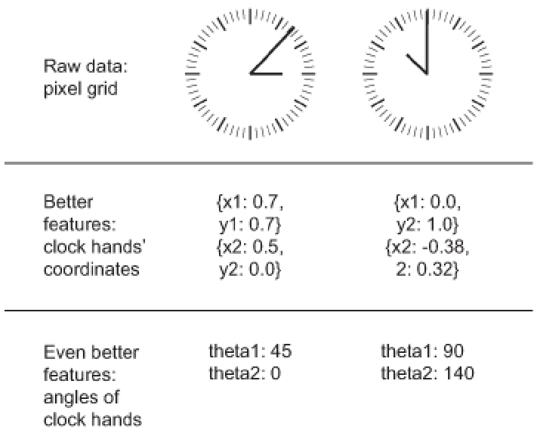
```
x -= x.mean(axis=0)
x /= x.std(axis=0)
```

Completing missing values

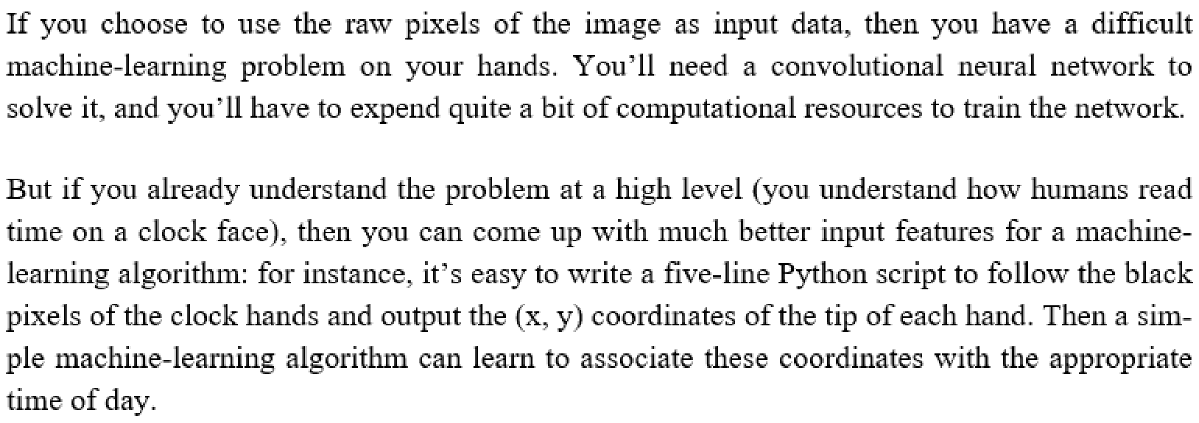
- In general, with neural networks, it’s safe to input missing values as 0, with the condition that 0 isn’t already a meaningful value.
- The network will learn from exposure to the data that the value 0 means missing data and will start ignoring the value.
- Note that if you’re expecting missing values in the test data, but the network was trained on data without any missing values, the network won’t have learned to ignore missing values!
- In this situation, you should artificially generate training samples with missing entries: copy some training samples several times, and drop some of the features that you expect are likely to be missing in the test data.

Feature Engineering

- *Feature engineering* is the process of using your own knowledge about the data and about the machine-learning algorithm at hand (in this case, a neural network) to make the algorithm work better by applying hardcoded (nonlearned) transformations to the data before it goes into the model.
- In many cases, it isn't reasonable to expect a machine-learning model to be able to learn from completely arbitrary data. The data needs to be presented to the model in a way that will make the model's job easier.



Raw data: pixel grid		
Better features: clock hands' coordinates	{x1: 0.7, y1: 0.7} {x2: 0.5, y2: 0.0}	{x1: 0.0, y2: 1.0} {x2: -0.38, 2: 0.32}
Even better features: angles of clock hands	theta1: 45 theta2: 0	theta1: 90 theta2: 140



If you choose to use the raw pixels of the image as input data, then you have a difficult machine-learning problem on your hands. You'll need a convolutional neural network to solve it, and you'll have to expend quite a bit of computational resources to train the network.

But if you already understand the problem at a high level (you understand how humans read time on a clock face), then you can come up with much better input features for a machine-learning algorithm: for instance, it's easy to write a five-line Python script to follow the black pixels of the clock hands and output the (x, y) coordinates of the tip of each hand. Then a simple machine-learning algorithm can learn to associate these coordinates with the appropriate time of day.

Feature Engineering

- The left side of Figure illustrates raw data from an input data source; the right side illustrates a **feature vector**, which is the set of floating-point values comprising the examples in your data set.
- **Feature engineering** means transforming raw data into a feature vector. Expect to spend significant time doing feature engineering.
- Many machine learning models must represent the features as real-numbered vectors since the feature values must be multiplied by the model weights.

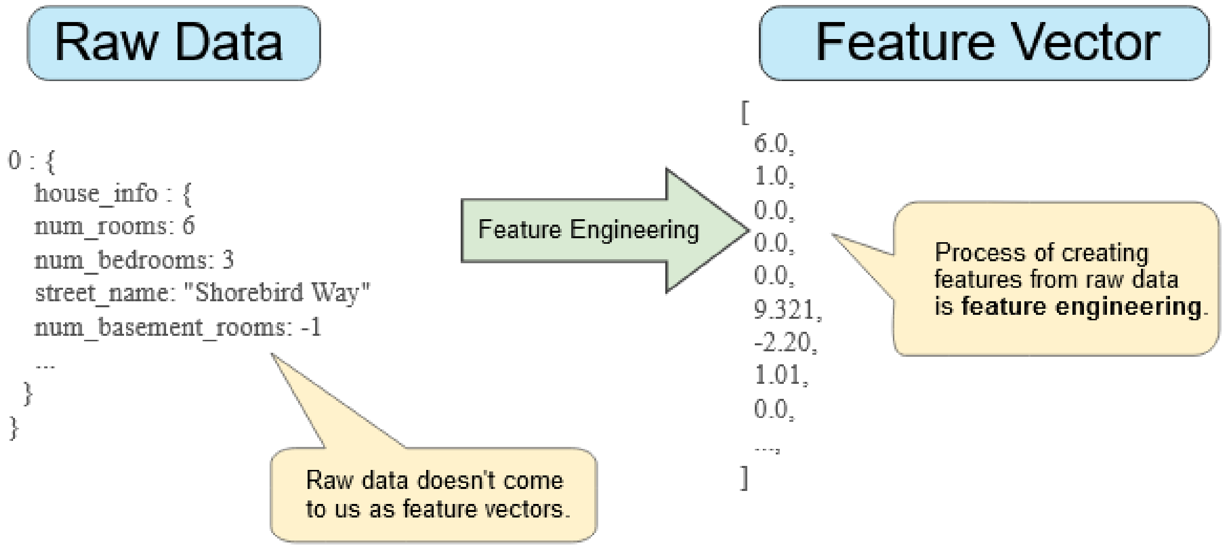


Figure 1. Feature engineering maps raw data to ML features.

https://developers.google.com/machine-learning/crash-course/representation/feature-engineering

Feature Engineering

- **Mapping categorical values**

- Categorical features have a discrete set of possible values. For example, there might be a feature called street_name with options that include:

  {'Charleston Road',
  'North Shoreline Boulevard',
  'Shorebird Way',
  'Rengstorff Avenue'}

- Since models cannot multiply strings by the learned weights, we use feature engineering to convert strings to numeric values.

- We can accomplish this by defining a mapping from the feature values, which we'll refer to as the vocabulary of possible values, to integers. Since not every street in the world will appear in our dataset, we can group all other streets into a catch-all "other" category, known as an OOV (out-of-vocabulary) bucket.

- map Charleston Road to **0**
- map North Shoreline Boulevard to **1**
- map Shorebird Way to **2**
- map Rengstorff Avenue to **3**
- map everything else (OOV) to **4**

Feature Engineering

- if a house is at the corner of two streets, then two binary values are set to 1, and the model uses both their respective weights.

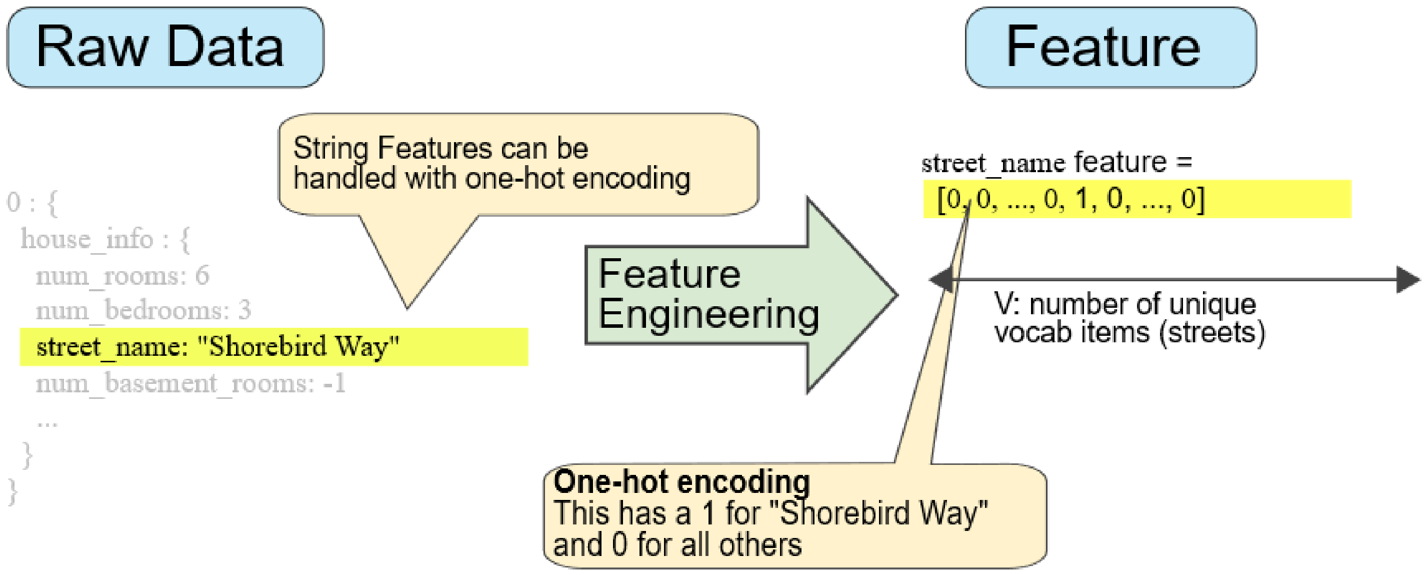
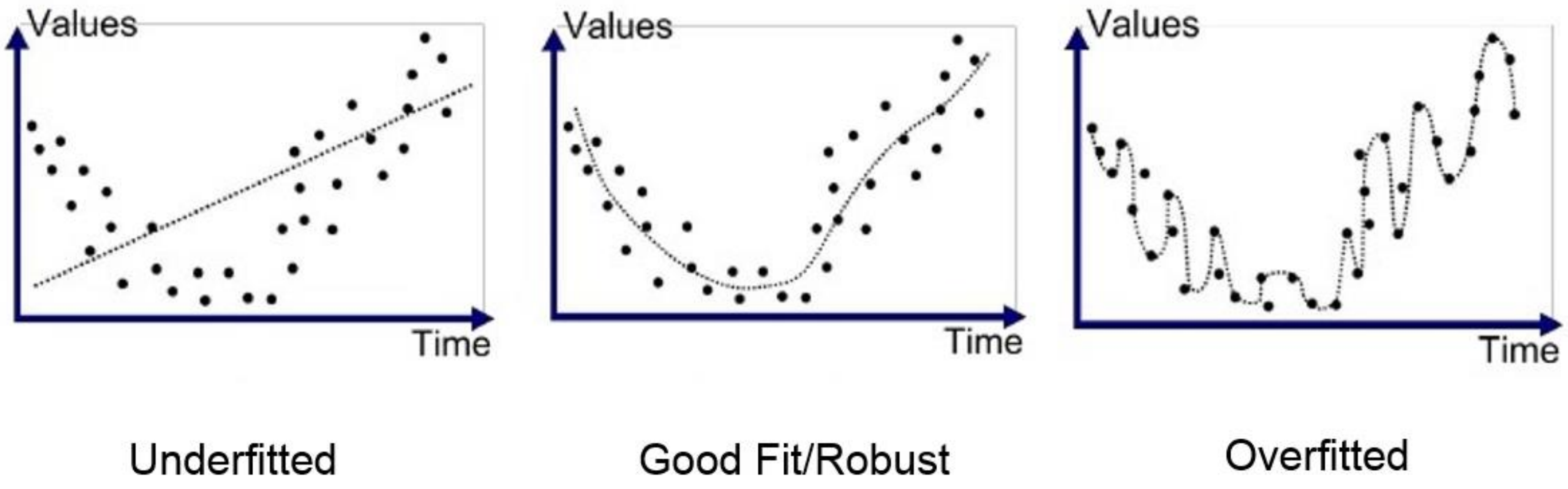

Raw Data
Feature
String Features can be handled with one-hot encoding
0 : {
house_info : {
num_rooms: 6
num_bedrooms: 3
street_name: "Shorebird Way"
num_basement_rooms: -1
...
}
}
Feature Engineering
street_name feature =
[0, 0, ..., 0, 1, 0, ..., 0]
V: number of unique vocab items (streets)
One-hot encoding
This has a 1 for "Shorebird Way" and 0 for all others

Figure 3. Mapping street address via one-hot encoding.

Underfitting and overfitting

- A model is said to be ***underfit***: there is still progress to be made; the network hasn't yet modeled all relevant patterns in the training data.
- But after a certain number of iterations on the training data, generalization stops improving, and validation metrics stall and then begin to degrade: the model is starting to **overfit**. That is, it's beginning to learn patterns that are specific to the training data but that are misleading or irrelevant when it comes to new data.

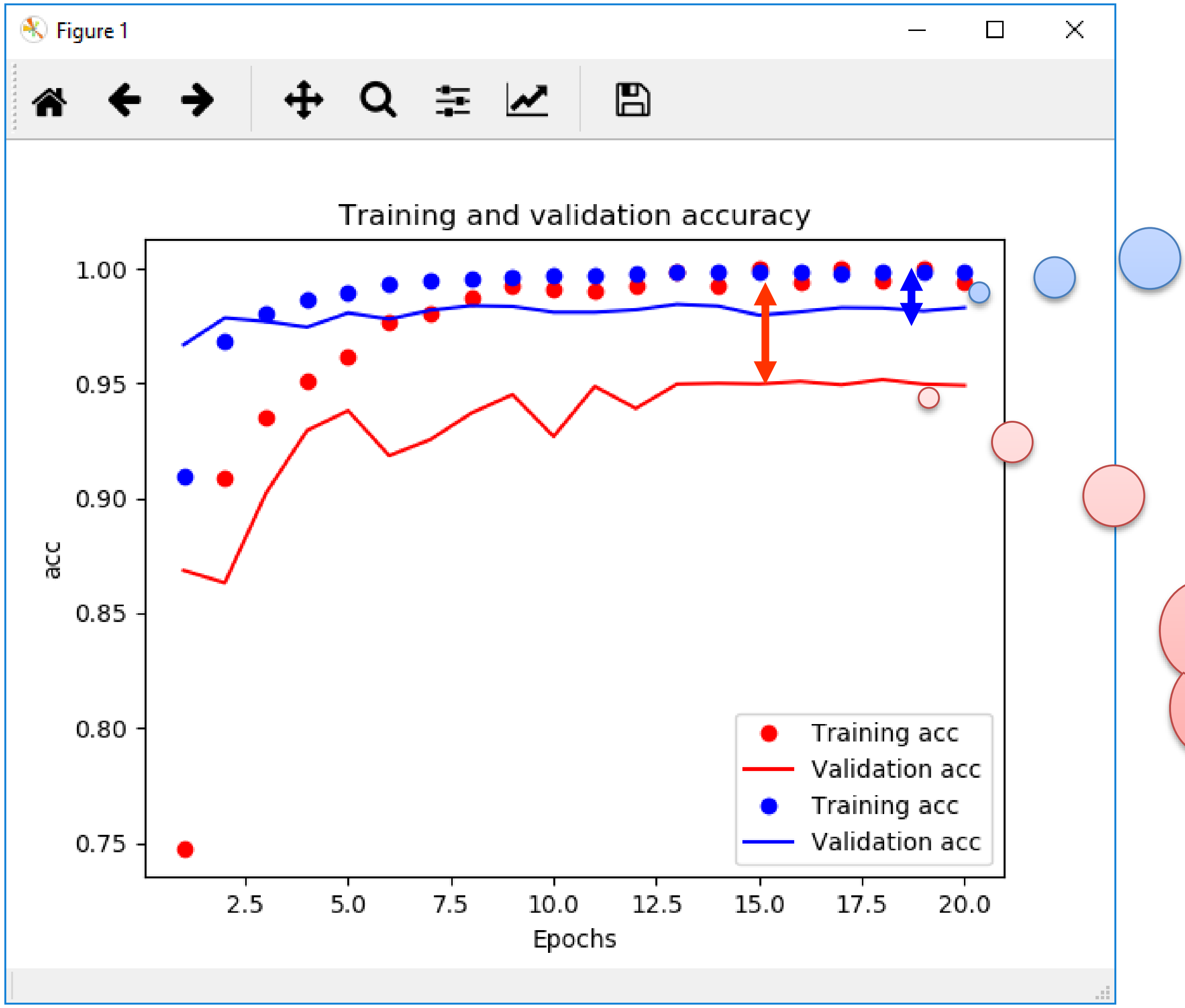


https://medium.com/greyatom/what-is-underfitting-and-overfitting-in-machine-learning-and-how-to-deal-with-it-6803a989c76

Regularization

- The processing of fighting overfitting this way is called regularization. Let’s review some of the most common regularization
  - Using more training data
  - Reducing network size
  - weight regularization
  - Dropout factor
  - Data augmentation

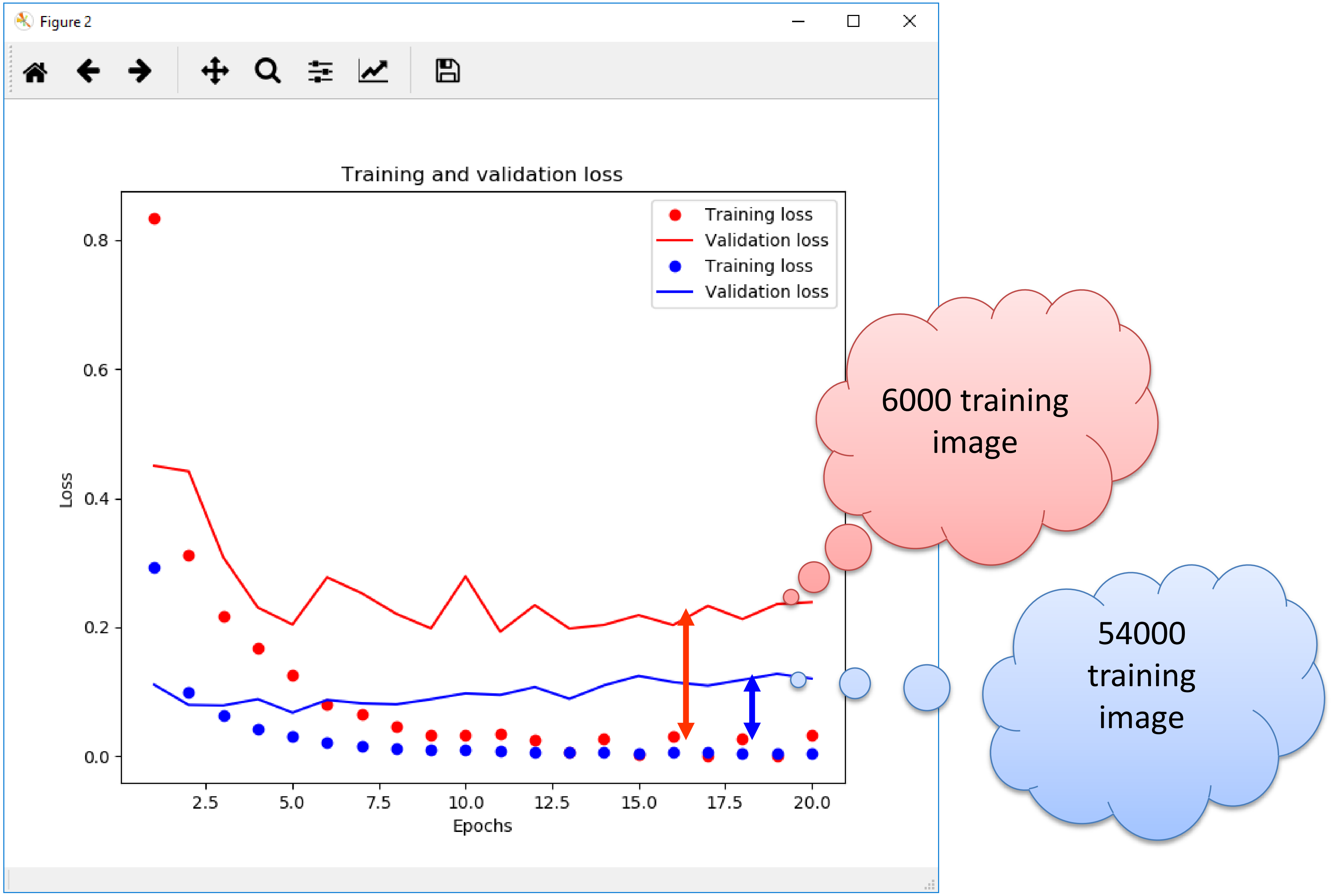
Regularization: Using more training data



Validation split=0.1 (54000 training image)

Validation split=0.9 (6000 training image)

Regularization: Using more training data



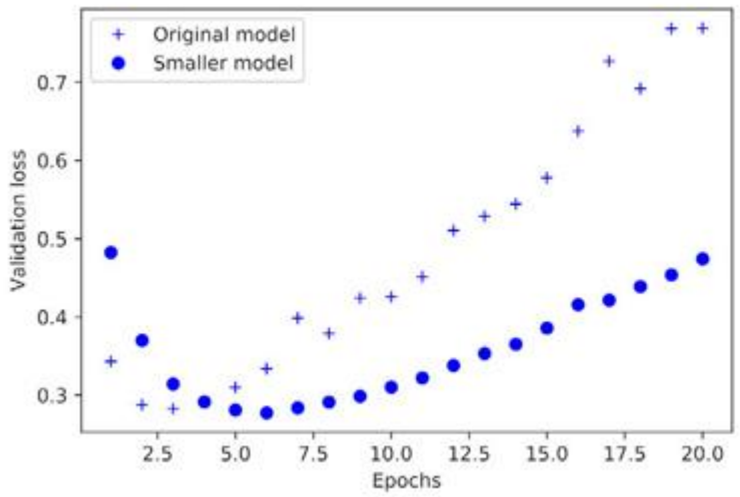
Regularization: Reducing network size

- The first step when dealing with overfitting is to decrease the complexity of the model.
- To decrease the complexity, we can simply remove layers or reduce the number of neurons to make the network smaller.
- While doing this, it is important to calculate the input and output dimensions of the various layers involved in the neural network.
- There is no general rule on how much to remove or how large your network should be.
- But, if your neural network is overfitting, try making it smaller.

the movie-review classification network.

```
from keras import models
from keras import layers

model = models.Sequential()
model.add(layers.Dense(16, activation='relu', input_shape=(10000,)))
model.add(layers.Dense(16, activation='relu'))
model.add(layers.Dense(1, activation='sigmoid'))
```

```
model = models.Sequential()
model.add(layers.Dense(4, activation='relu', input_shape=(10000,)))
model.add(layers.Dense(4, activation='relu'))
model.add(layers.Dense(1, activation='sigmoid'))
```



Regularization: Weight regularization

- A common way to mitigate overfitting is to put constraints on the complexity of a network by forcing its weights to take only small values, which makes the distribution of weight values more regular.
- This is called weight regularization, and it's done by adding to the loss function of the network a cost associated with having large weights. This cost comes in two flavors:
- L1 regularization—The cost added is proportional to the absolute value of the weight coefficients (the L1 norm of the weights).
- L2 regularization—The cost added is proportional to the square of the value of the weight coefficients (the L2 norm of the weights). L2 regularization is also called weight decay in the context of neural networks. Don't let the different name confuse you: weight decay is mathematically the same as L2 regularization

$$Loss = Error(y, \hat{y})$$

Loss function with no regularisation

$$Loss = Error(y, \hat{y}) + \lambda \sum_{i=1}^{N} |w_i|$$

Loss function with L1 regularisation

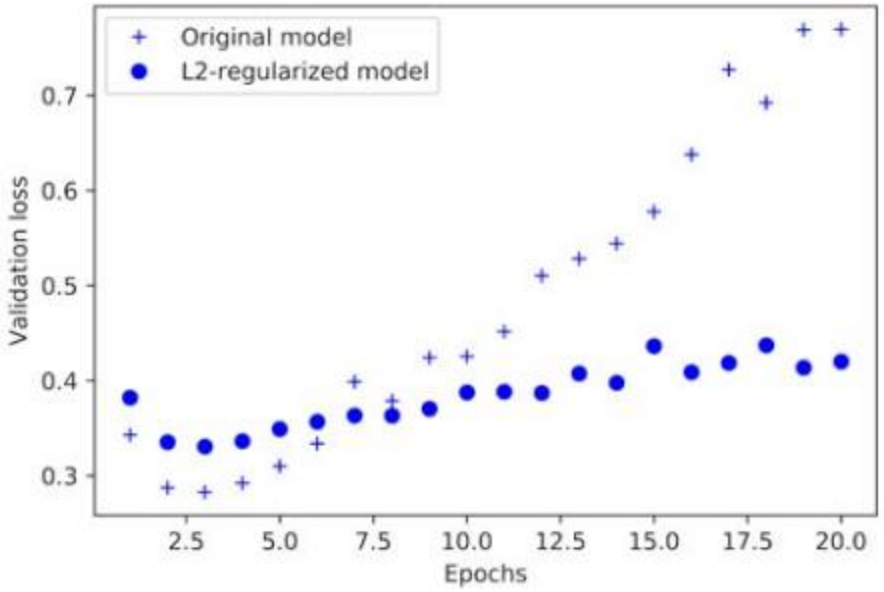
$$Loss = Error(y, \hat{y}) + \lambda \sum_{i=1}^{N} w_i^2$$

Loss function with L2 regularisation

https://towardsdatascience.com/intuitions-on-l1-and-l2-regularisation-235f2db4c261

Regularization: Weight regularization

```
from keras import regularizers

model = models.Sequential()
model.add(layers.Dense(16, kernel_regularizer=regularizers.l2(0.001),
                       activation='relu', input_shape=(10000,)))
model.add(layers.Dense(16, kernel_regularizer=regularizers.l2(0.001),
                       activation='relu'))
model.add(layers.Dense(1, activation='sigmoid'))
```



```
from keras import regularizers

regularizers.l1(0.001)              <—— L1 regularization

regularizers.l1_l2(l1=0.001, l2=0.001)   <—┘ Simultaneous L1 and
                                             L2 regularization
```

Regularization: Dropout regularization

- The key idea is to randomly drop units (along with their connections) from the neural network during training. This prevents units from co-adapting too much. During training, dropout samples from an exponential number of different "thinned" networks. At test time,it is easy to approximate the effect of averaging the predictions of all these thinned networks by simply using a single unthinned network that has smaller weights. This significantly reduces overfitting and gives major improvements over other regularization methods.

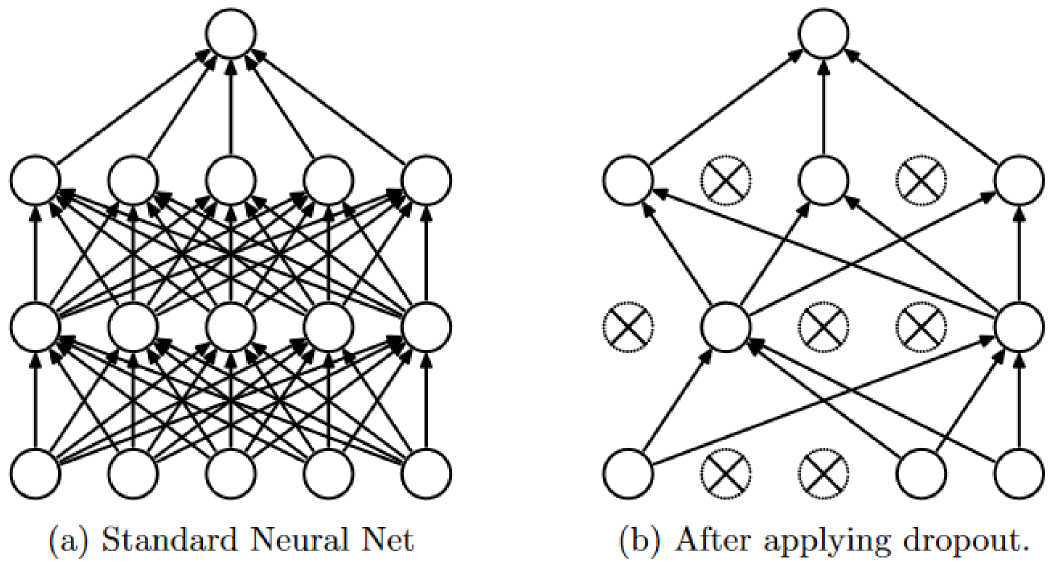
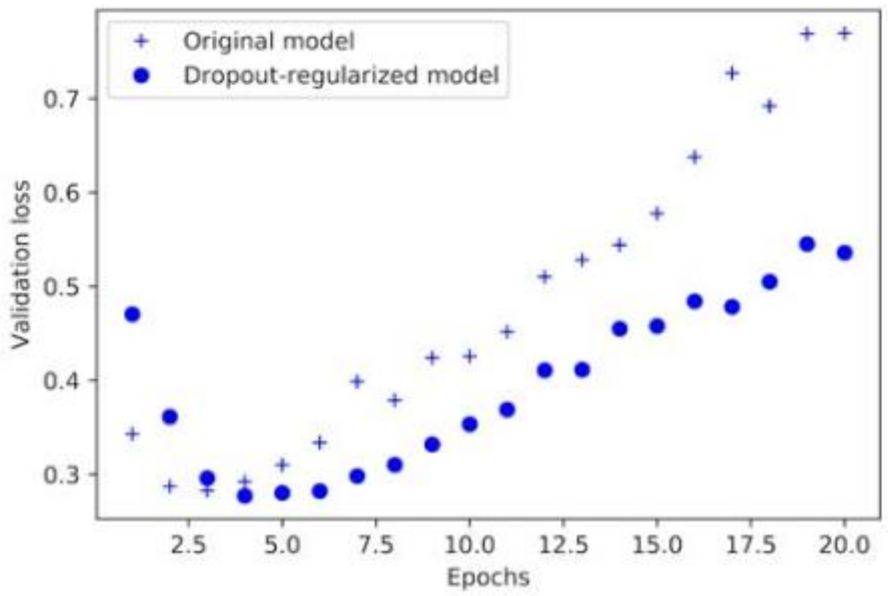


Figure 1: Dropout Neural Net Model. **Left**: A standard neural net with 2 hidden layers. **Right**: An example of a thinned net produced by applying dropout to the network on the left. Crossed units have been dropped.

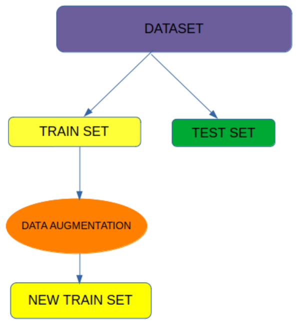
https://jmlr.org/papers/volume15/srivastava14a.old/srivastava14a.pdf

Regularization: Dropout regularization

```
model = models.Sequential()
model.add(layers.Dense(16, activation='relu', input_shape=(10000,)))
model.add(layers.Dropout(0.5))
model.add(layers.Dense(16, activation='relu'))
model.add(layers.Dropout(0.5))
model.add(layers.Dense(1, activation='sigmoid'))
```

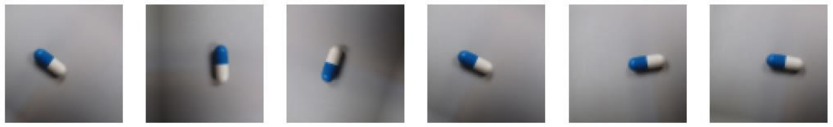


Regularization: Data augmentation

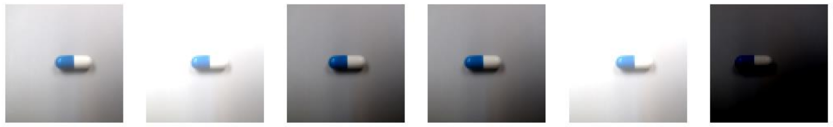


https://neptune.ai/blog/data-augmentation-nlp

- In computer vision applications data augmentations are done almost everywhere to get larger training data and make the model generalize better. The main methods used involve: cropping, flipping, zooming, rotation, noise injection, and many others.

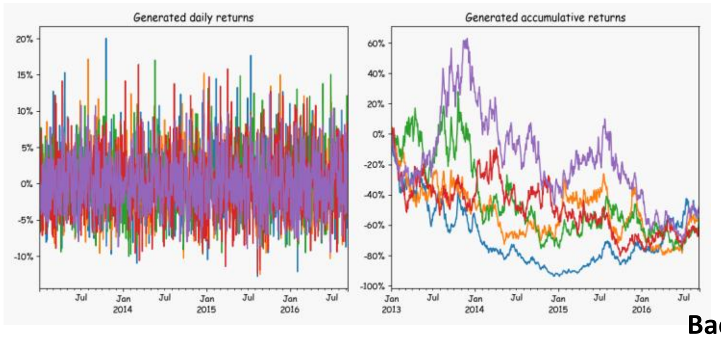


RandomRotate



RandomLighting

https://becominghuman.ai/data-augmentation-using-fastai-aefa88ca03f1



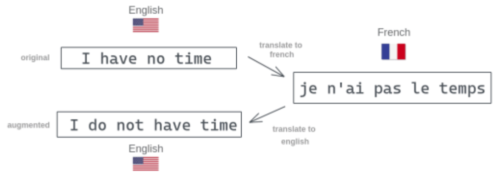
Data augmentation to generate new financial series.
Generally, our main goal is to develop algorithms that model financial series in order to get investment rules. Nevertheless, sometimes there isn't so much historical data as desired, so it would be pretty useful to widen the data.

https://quantdare.com/variational-autoencoder-as-a-method-of-data-augmentation/

Some NLP data augmentation methods:

- Back translation.
- EDA (Easy Data Augmentation).
- NLP Albumentation.

Back translation: In this method, we translate the text data to some language and then translate it back to the original language.



Easy Data Augmentation

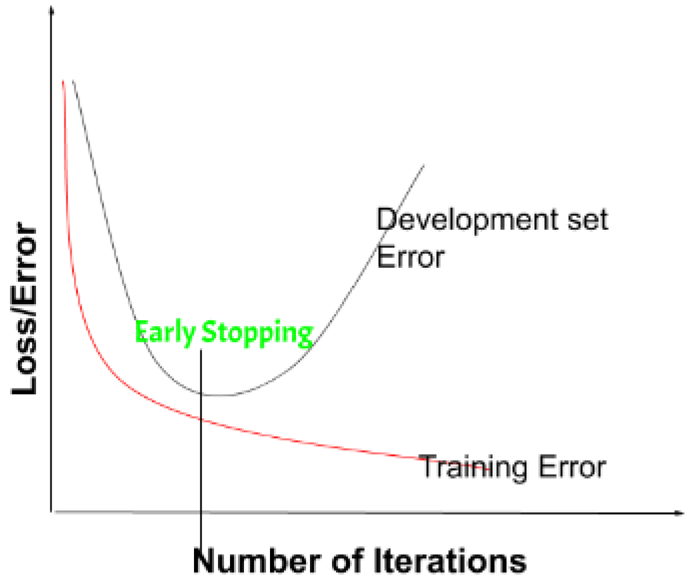
*This **article** will focus on summarizing data augmentation **techniques** in NLP.*
This **write-up** will focus on summarizing data augmentation **methods** in NLP.

NLP Albumentation

•**Shuffle Sentences Transform**: In this transformation, if the given text sample contains multiple sentences these sentences are shuffled to create a new sample. For example:
text = '<Sentence1>. <Sentence2>. <Sentence4>. <Sentence4>. <Sentence5>. <Sentence5>.'
Is transformed to:
text = '<Sentence2>. <Sentence3>. <Sentence1>. <Sentence5>. <Sentence5>. <Sentence4>.'

Regularization: EarlyStopping

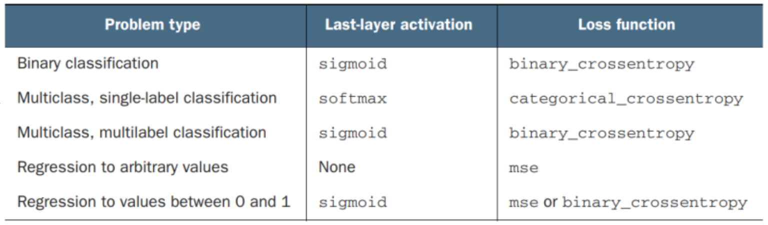
- Here, the second case is analyzed. In early stopping, the algorithm is trained using the training set and the point at which to stop training is determined from the validation set. Training error and validation error are analysed.
- The training error steadily decreases while validation error decreases until a point, after which it increases. This is because, during training, the learning model starts to overfit to the training data. This causes the training error to decrease while the validation error increases.
- So a model with better validation set error can be obtained if the parameters that give the least validation set error are used.



earlyStopping =EarlyStopping(monitor='val_loss', patience=30, verbose=0, mode='min')

The universal workflow of machine learning

- ***Defining the problem and assembling a dataset***
- ***Choosing a measure of success***
  Deciding on an evaluation protocol
- ***Preparing your data***
- ***Developing a model that does better than a baseline***
  - *Last-layer activation*
  - *Loss function*
  - Optimization configuration
- ***Scaling up: developing a model that overfits***
  - Add layers.
  - Make the layers bigger.
  - Train for more epochs,
- ***Regularizing your model and tuning your hyperparameters***
  - Add dropout.
  - Try different architectures: add or remove layers.
  - Add L1 and/or L2 regularization.
  - Try different hyperparameters (such as the number of units per layer or the learning rate of the optimizer) to find the optimal configuration.
  - Optionally, iterate on feature engineering: add new features, or remove features that don't seem to be informative.



Problem type	Last-layer activation	Loss function
Binary classification	sigmoid	binary_crossentropy
Multiclass, single-label classification	softmax	categorical_crossentropy
Multiclass, multilabel classification	sigmoid	binary_crossentropy
Regression to arbitrary values	None	mse
Regression to values between 0 and 1	sigmoid	mse or binary_crossentropy