Data representations - tensors

```
>>> import numpy as np
>>> x = np.array(12)
>>> x
array(12)
>>> x.ndim
0
```

```
>>> x = np.array([12, 3, 6, 14])
>>> x
array([12, 3, 6, 14])
>>> x.ndim
1
```

```
>>> x = np.array([[5, 78, 2, 34, 0],
[6, 79, 3, 35, 1],
[7, 80, 4, 36, 2]])
>>> x.ndim
2
```

Key attributes

Number of axes (rank)

Shape

Data type

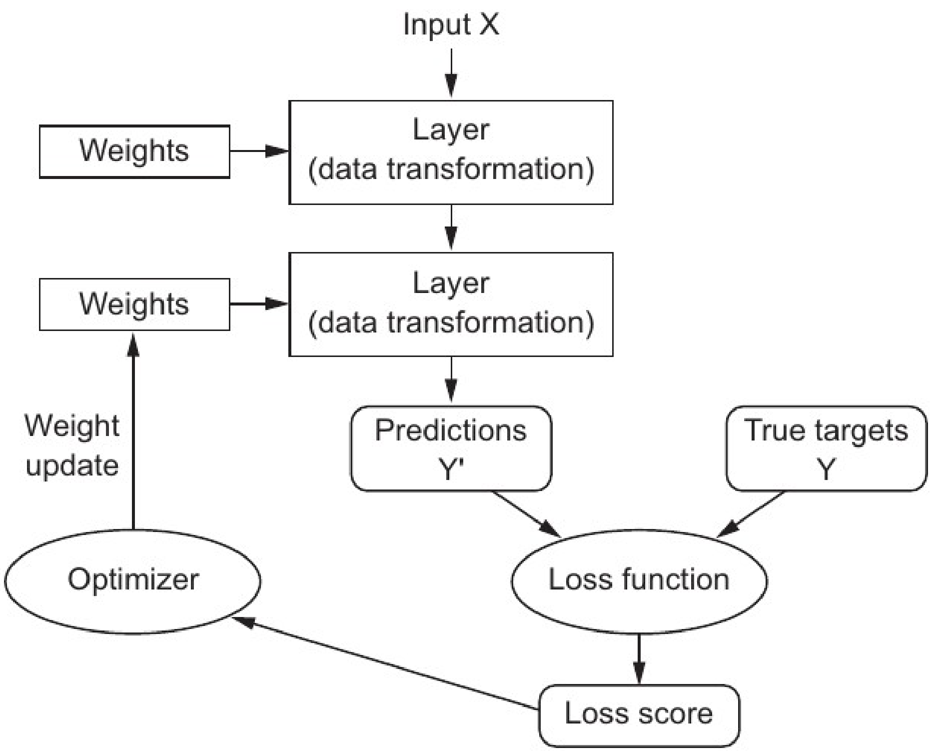
Data representations - tensors

Real-world examples of data tensors

 Vector data—2D tensors of shape (samples, features)
 Timeseries data or sequence data—3D tensors of shape (samples, timesteps, features)
 Images—4D tensors of shape (samples, height, width, channels) or (samples, channels, height, width)
 Video—5D tensors of shape (samples, frames, height, width, channels) or (samples, frames, channels, height, width)

Network

- Layers, which are combined into a network (or model)
- The input data and corresponding targets
- The loss function, which defines the feedback signal used for learning
- The optimizer, which determines how learning proceeds



Network

Different layers are appropriate for different tensor formats and different types of data Processing:

- simple vector data, stored in 2D tensors of shape (samples, features), is processed by densely connected layers (the Dense class in Keras);
- sequence data, stored in 3D tensors of shape (samples,timesteps, features) , is processed by recurrent layers such as an LSTM layer;
- Image data, stored in 4D tensors, is processed by 2D convolution layers (Conv2D).

Label encoding

```
#One hot encoding – on output we get probabilities, better
interpretability

Dimension = 46
Label = 34
res = np.asarray([label==i for i in
range(0,dimension)],dtype=np.uint8)
print(res)

[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 1 0 0
 0 0 0 0 0 0 0 0 0]
```

Label encoding

```
#Integer encoding

y_train = np.array(train_labels)
y_test = np.array(test_labels)


model.compile(optimizer='rmsprop',
loss='sparse_categorical_crossentropy',
metrics=['acc'])
```

Label encoding

Integer encoding assumes that „difference” between classes „1” and „2” is the same as between „2” and „3”

In most practical cases it does not make any sense.

For these reasons use one-hot encoding to code categorical data like class labels.

Cross-validation

- Must code by hand in Keras

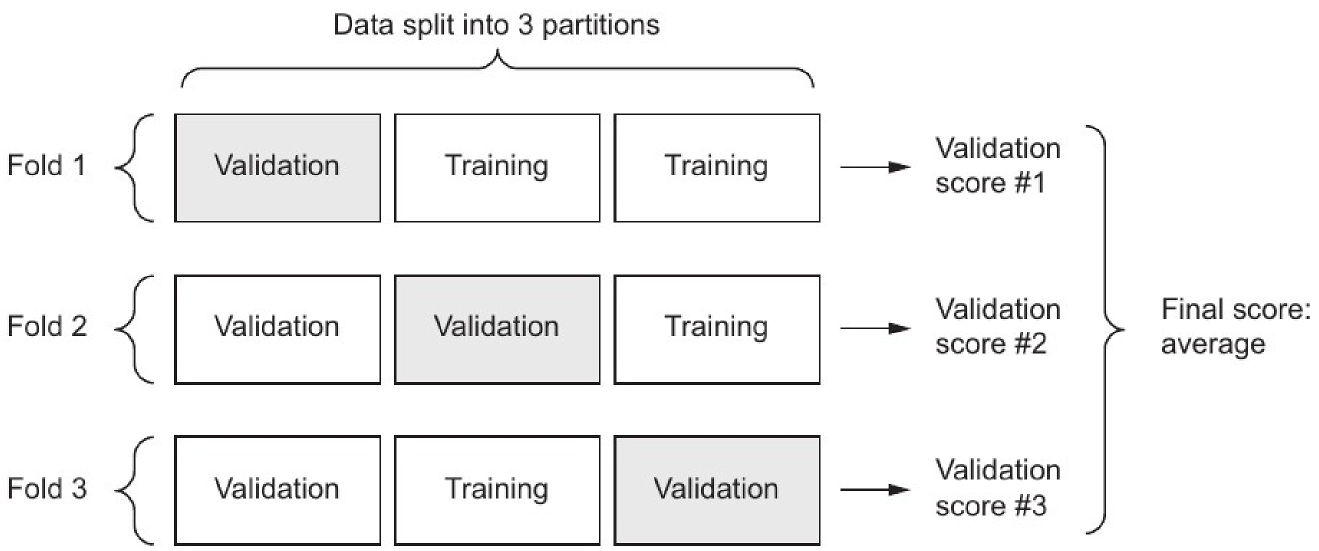


Figure 3.11 3-fold cross-validation

Cross-validation

Prepares the validation data: data from partition #k

Prepares the training data: data from all other partitions

```
for i in range(k):
    print('processing fold #', i)
    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]

    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)

    model = build_model()
    model.fit(partial_train_data, partial_train_targets,
              epochs=num_epochs, batch_size=1, verbose=0)
    val_mse, val_mae = model.evaluate(val_data, val_targets, verbose=0)
    all_scores.append(val_mae)
```

Builds the Keras model (already compiled)

Trains the model (in silent mode, verbose = 0)

Evaluates the model on the validation data

Data preprocessing

VECTORIZATION

Data must be first turn into tensors

VALUE NORMALIZATION – USE ONLY TRAINING SET, APPLY TO ALL DATA!!!

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.

HANDLING MISSING VALUES

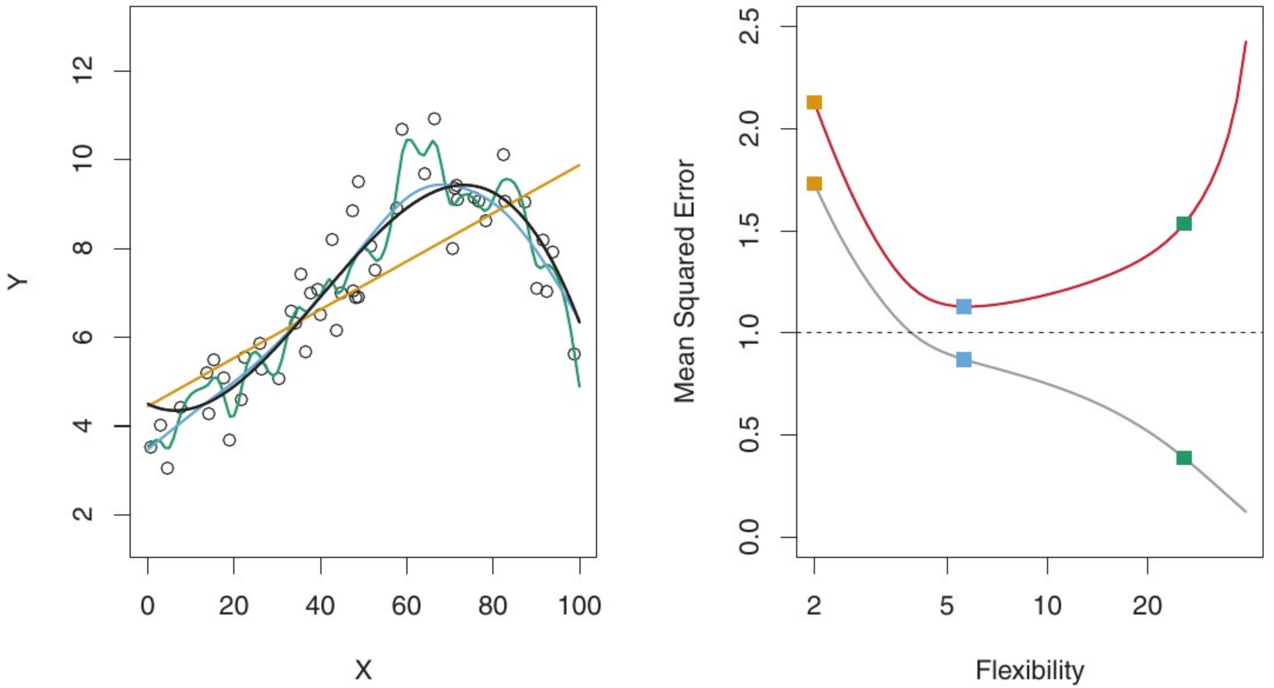
it's safe to input missing values as any value which is not meaningful

Feature engineering vs. feature learning

Good features still allow you to solve problems more elegantly.

Good features let you solve a problem with far less data. The ability of deep-learning models to learn features on their own relies on having lots of training data available; if you have only a few samples, then the information value in their features becomes critical.

Bias-Variance trade-off: underfitting or overfitting



Left: Data simulated from f , shown in black. Three estimates of f are shown: the linear regression line (orange curve), and two smoothing spline fits (blue and green curves). Right: Training MSE (grey curve), test MSE (red curve), and minimum possible test MSE over all methods (dashed line). Squares represent the training and test MSEs for the three fits shown in the left-hand panel.

Preventing overfitting

Data-related prevention:

- Get more data
- Use data augmentation

Preventing overfitting

Architecture selection – use model with lower capacity

Listing 1 Original model

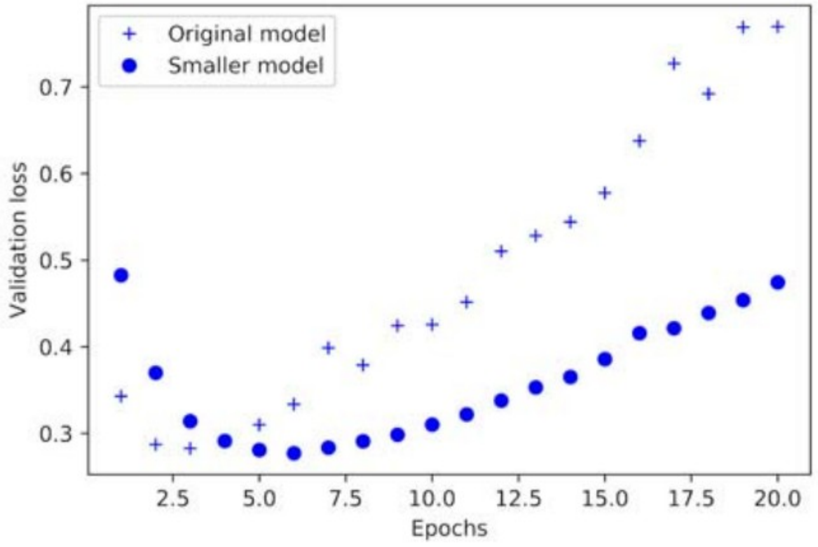
```
from keras import models
from keras import layers
model = models.Sequential()
model.add(layers.Dense(16, activation='relu', input_shape=(10000,)))
model.add(layers.Dense(16, activation='relu'))
model.add(layers.Dense(1, activation='sigmoid'))
```

Listing 2 Version of the model with lower capacity

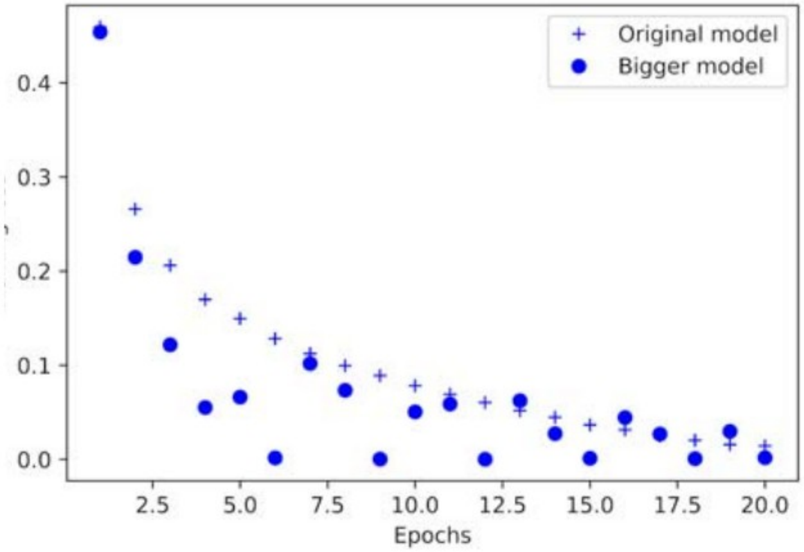
```
model = models.Sequential()
model.add(layers.Dense(4, activation='relu', input_shape=(10000,)))
model.add(layers.Dense(4, activation='relu'))
model.add(layers.Dense(1, activation='sigmoid'))
```

Preventing overfitting

Architecture selection



Effect of model
capacity on validation loss



Effect of model
capacity on training loss

Preventing overfitting

Regularization

$$L = \frac{1}{N}\sum_{i=1}^{N} L_i + \lambda R(W)$$

$$R(W) = \sum_i \sum_j W_{i,j}^2$$

$$R(W) = \sum_i \sum_j |W_{i,j}|$$

$$R(W) = \sum_i \sum_j \beta W_{i,j}^2 + |W_{i,j}|$$

$$L = \frac{1}{N}\sum_{i=1}^{N}[-log(e^{s_{y_i}} / \sum_j e^{s_j})] + \lambda \sum_i \sum_j W_{i,j}^2$$

$$L = \frac{1}{N}\sum_{i=1}^{N}\sum_{j \neq y_i}[max(0, s_j - s_{y_i} + 1)] + \lambda \sum_i \sum_j W_{i,j}^2$$

$$W = W - \alpha \nabla_W f(W) - \lambda R(W)$$

Preventing overfitting

Regularization

Listing 1 Adding L2 weight regularization to the model

```
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'))
```

Preventing overfitting

Regularization

Listing 4.7 Different weight regularizers available in Keras

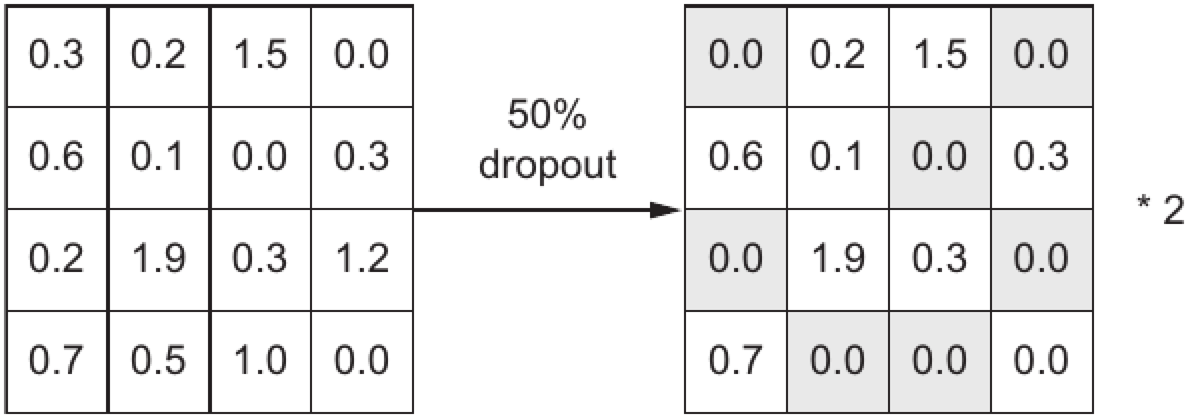
```
from keras import regularizers

regularizers.l1(0.001)                 <-- L1 regularization

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

Preventing overfitting

Dropout



Dropout applied to an activation matrix at training time, with rescaling happening during training. At test time, the activation matrix is unchanged.

Preventing overfitting

Dropout

Listing 1 Adding dropout to a network

```
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'))
```

Overfitting – is it really bad?

The ideal model is one that stands right at the border between underfitting and overfitting.

To figure out where this border lies, first you must cross it.

To figure out how big a model you'll need, you must develop a model that overfits.
This is fairly easy:
1 Add layers.
2 Make the layers bigger.
3 Train for more epochs.

The next stage is to start regularizing and tuning the model, to get as close as possible to the ideal model that neither underfits nor overfits.