機器學習研究應用
Study for Machine Learning and Its Applications

Fundamentals of Machine Learning

孫士韋
Shih-Wei Sun
swsun@newmedia.tnua.edu.tw

Outline

- Four branches of machine learning
- Evaluating machine-learning models
- Data preprocessing, feature engineering, and feature learning
- Overfitting and underfitting
- The universal workflow of machine learning

4 Branches of machine learning (1/2)

- Supervised learning (goal of this course)
  - Binary classification: IMDB: like, dislike
  - multiclass classification: Reuters news, 46 classes
  - Scalar regression: Boston housing price
  - Learn the training input / training targets (label, annotation)
- Examples:
  - Optical character recognition, Speech recognition
  - Image classification, Language translation
  - Sequence generation: given a image, predict a caption
  - Object detection: given a picture, draw a bounding box
  - Image segmentation: given a picture, draw a mask

chair

Car,
[Sun et al., JVCI, 13]

Segmentation

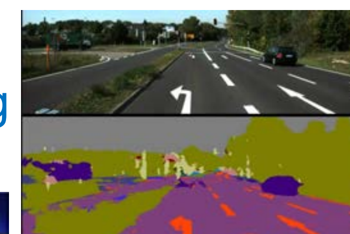
4 Branches of machine learning (2/2)

- Supervised learning (major part)
- Unsupervised learning

Image compression

  - Data visualization, compression, denoising
    - Dimensionality reduction, clustering (math field)
- Self-supervised learning
  - Supervised learning: without human-annotated labels
  - labels: generated from the input data, autoencoder
    - Predict the next frame in a video / given past frames
    - Predict the next word in a text / given previous words

Self-driving car



- Reinforcement learning

  - Google DeepMind, Alpha Go, agent
    - Receives information about its environment
    - Learns to choose actions, maximize some reward

robot

Evaluating machine-learning models

- Generalize the model
  - Never-before-seen data
- Training, validation, test sets
  - Splitting the available data into 3 sets
  - Avoid information leaks
    - Training, validation data: similar manner
  - Test dataset: never-before-seen
    - New data, ex: interactive hand gesture, talk to Siri

Validations

- Simple hold-out validation: early examples

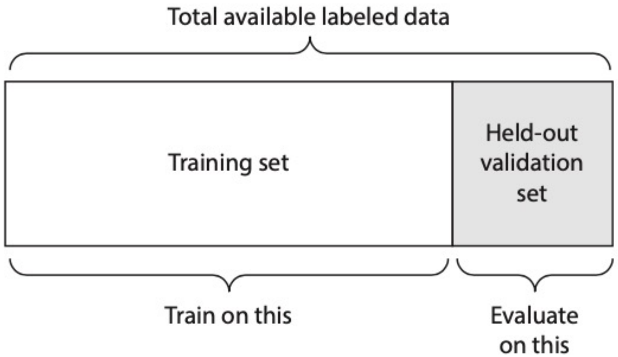


Figure 4.1 Simple hold-out validation split

```
my_slice = train_images[10:100, :, :]
print(my_slice.shape)

my_slice = train_images[10:100, 0:28, 0:28]
print(my_slice.shape)
```

- K-fold validation: previous example (last week)

```
for i in range(k):
    print('processing fold #%d' % 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)
```

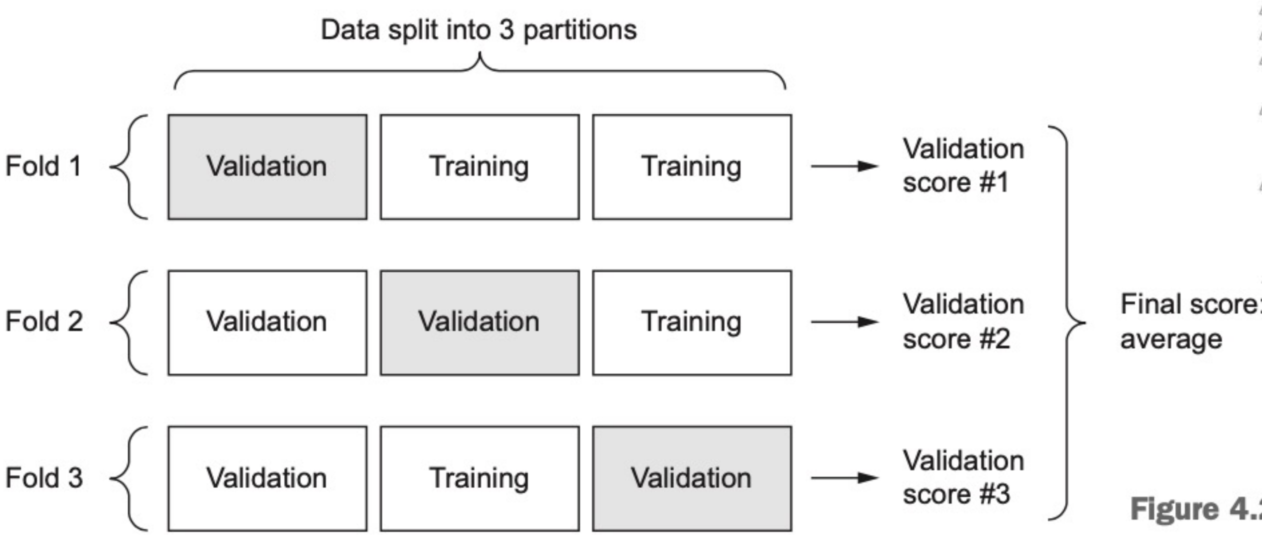


Figure 4.2 Three-fold validation

Things to keep in mind

- Data representativeness
  - In a digit recognition example
    - First 80%: training set
    - Remaining 20%: testing set
    - Model: 0-7
    - Test for 8,9: ridiculous!
  - Random shuffle the samples!
- Arrow of time
  - Predict the future / given the past
  - Should not shuffle your data
- Redundancy: data points appear twice, avoid

Data preprocessing for neural networks (1/2)

- Vectorization
  - Must be Tensors of float point data
    - Sound, image: float32 data
    - Text: sequence of words: one-hot encoding
- Value normalization
  - Image data: 0 – 255 range (gray scale values)
    - Divide by 255: values in 0 – 1 range
  - House price:
    - Standard deviation of 1 and mean of 0

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

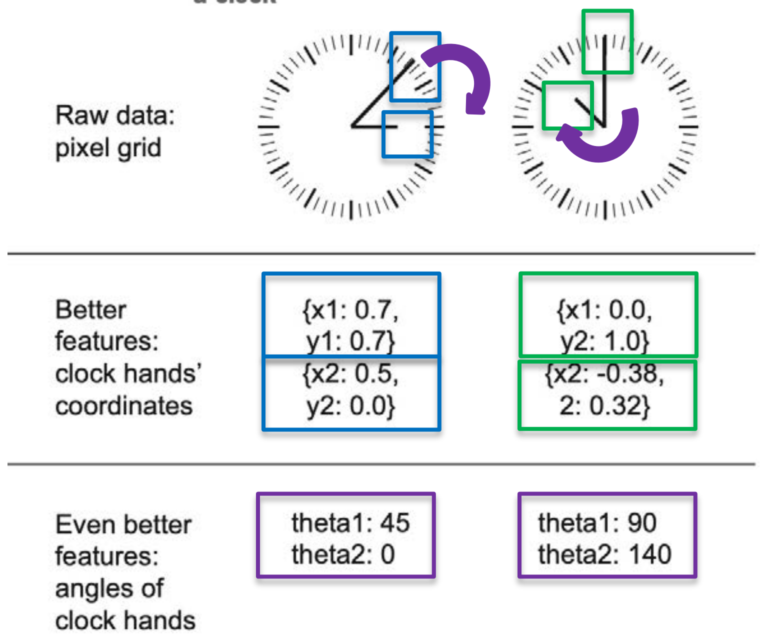
Assuming x is a 2D data matrix of shape (samples, features)

Data preprocessing for neural networks (2/2)

Feature engineering: Critical role!!
-Make to problem solved or not
-Far less data

- Handling missing values
  - Sometimes missing some data
    - Set it as 0
      - 0: means missing data, start ignoring the value
    - Copy some training samples
- Feature engineering
  - Input an image
  - Output the time of a day
  - Input
    - Raw pixels: a bad way
    - Feature values: better

Figure 4.3 Feature engineering for reading the time on a clock



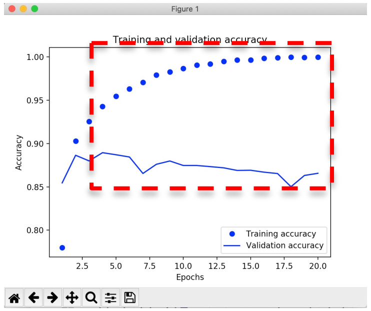
In the previous examples

- Overfit happens in every
  - Machine learning problem
- Optimization

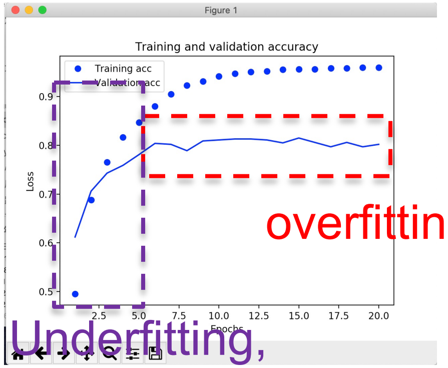
  The thing we can control
  - Adjusting a model
    - To get the best performance
- Generalization
  - How well of the performance
  - Target: get good generalization
- The best solution for overfitting
  - Get more training data!

  Get naturally generalized. But…We don't have it often.

IMDB binary classification, accuracy



Reuters Newswire, multiclass classification, accuracy



overfitting

Underfitting,
more epochs,
more learned parameters,
more accuracy

Skills to Fight Overfitting

- The overfitting problem:
  - The model memorized too many things
    - From the training data
  - Try to remove the too many memorized things
- To make the model
  - More generalized
- 3 methods:
  - Reducing the network's size
  - Adding weight regularization
  - Adding dropout

Method 1:
Reducing the network's size

- The simplest way to prevent overfitting
  - reduce number of learnable parameters in the model: 16 -> 4
  - Reduce the memorization capacity

Listing 4.3 Original model

No magical formula to determine: right number

Try to start from a small number

```
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 4.4 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'))
```

Starting from an IMDB Example (1/2)

- Load the data / label
- Transform texts to integers

```python
from tensorflow.keras.datasets import imdb
import numpy as np

(train_data, train_labels), (test_data, test_labels) = imdb.load_data(num_words=10000)

def vectorize_sequences(sequences, dimension=10000):
    # Create an all-zero matrix of shape (len(sequences), dimension)
    results = np.zeros((len(sequences), dimension))
    for i, sequence in enumerate(sequences):
        results[i, sequence] = 1.  # set specific indices of results[i] to 1s
    return results

```

Load the data, label

Transform texts to integers

Starting from an IMDB Example (2/2)

- Preparing the data

```
# Our vectorized training data
x_train = vectorize_sequences(train_data)
# Our vectorized test data
x_test = vectorize_sequences(test_data)
# Our vectorized labels
y_train = np.asarray(train_labels).astype('float32')
y_test = np.asarray(test_labels).astype('float32')

```

Call the function to
Transform texts to integers

labels
to
integers

Setting up for the Neural Network

- Step 1: add
- Step 2: compile
- Step 3: fit

```python
from tensorflow.keras import models
from tensorflow.keras import layers

#reducing the network's size
original_model = models.Sequential()
original_model.add(layers.Dense(16, activation='relu', input_shape=(10000,)))
original_model.add(layers.Dense(16, activation='relu'))
original_model.add(layers.Dense(1, activation='sigmoid'))
original_model.compile(optimizer='rmsprop', loss='binary_crossentropy', metrics=['acc'])

original_hist = original_model.fit(x_train, y_train, epochs=20, batch_size=512,
    validation_data=(x_test, y_test))
```

importing models, layers

Step1: add layers

Step 2: Compile, Setup the configuration

Step 3: fit the model

Plot the Overfitting Curve

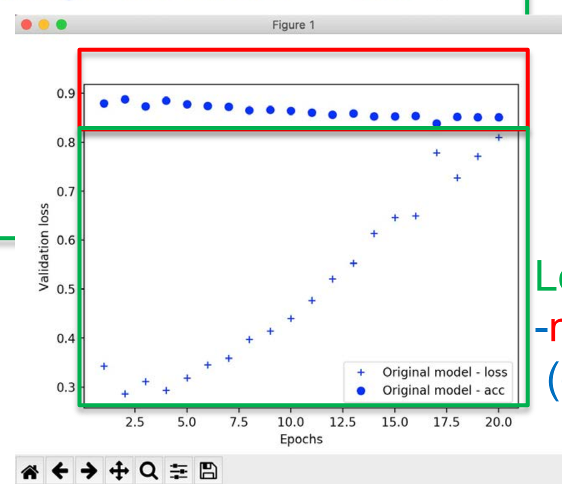
- Plot the results

```python
epochs = range(1, 21)
original_val_loss = original_hist.history['val_loss']

original_val_acc = original_hist.history['val_acc']

import matplotlib.pyplot as plt

# b+ is for "blue cross"
plt.plot(epochs, original_val_loss, 'b+', label='Original model - loss')

plt.plot(epochs, original_val_acc, 'bo', label='Original model - acc')

plt.xlabel('Epochs')
plt.ylabel('Validation loss')
plt.legend()
plt.show()
```

Assign Validation loss, acc

Plot the results

```
Epoch 3/20
25000/25000 [==============================] - 2s 77us/step - loss: 0.2619 - acc
: 0.9214 - val_loss: 0.3364 - val_acc: 0.8842
Epoch 4/20
25000/25000 [==============================] - 3s 112us/step - loss: 0.2407 - ac
c: 0.9307 - val_loss: 0.3681 - val_acc: 0.8719
Epoch 5/20
25000/25000 [==============================] - 3s 114us/step - loss: 0.2286 - ac
c: 0.9364 - val_loss: 0.3667 - val_acc: 0.8745
Epoch 6/20
25000/25000 [==============================] - 2s 88us/step - loss: 0.2212 - acc
: 0.9392 - val_loss: 0.3767 - val_acc: 0.8718
Epoch 7/20
22016/25000 [=========================>....] - ETA: 0s - loss: 0.2164 - acc: 0.9
422
```

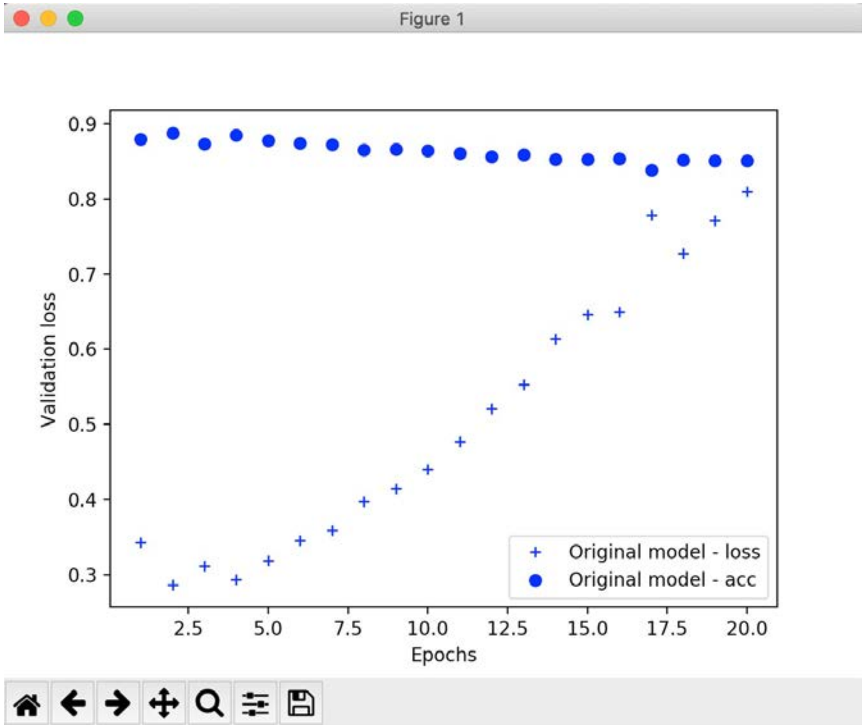


Accuracy unstable, decreased

Loss increased -not decrease (good way),

Practice 1

- Plot the overfitting curve



Method 1:
Reducing the network's size

- Network size: 16 -> 4

Step1: add layers,
Smaller network's size
16 -> 4

Step 2: Compile,
Setup the
configuration

```
#method 1: reducing the network's size
smaller_model = models.Sequential()
smaller_model.add(layers.Dense(4, activation='relu', input_shape=(10000,)))
smaller_model.add(layers.Dense(4, activation='relu'))
smaller_model.add(layers.Dense(1, activation='sigmoid'))
smaller_model.compile(optimizer='rmsprop', loss='binary_crossentropy', metrics=['acc'])

smaller_model_hist = smaller_model.fit(x_train, y_train, epochs=20, batch_size=512,
    validation_data=(x_test, y_test))

smaller_model_val_loss = smaller_model_hist.history['val_loss']
```

Step 3: Fit the model

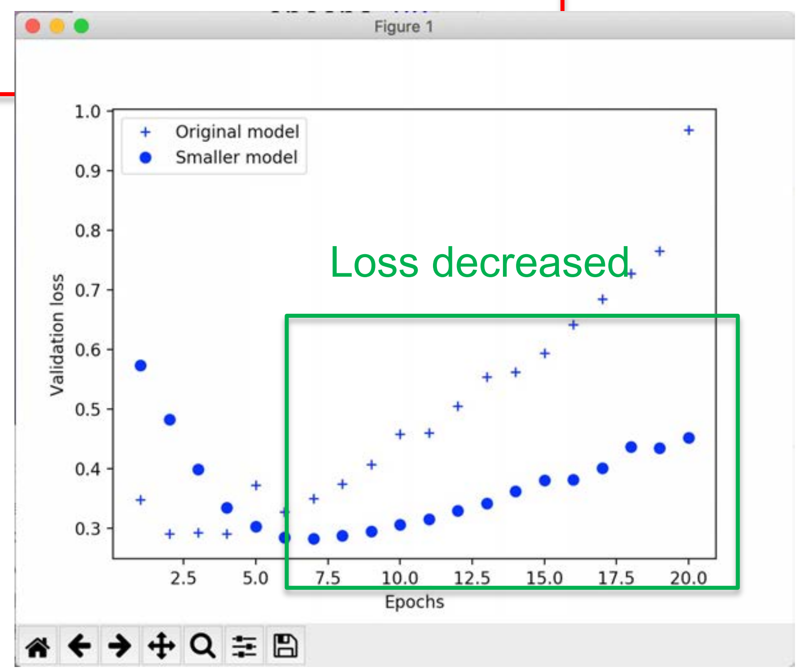
Assigning the loss results

Plot the network reducing results

- Plot the results

```python

# b+ is for "blue cross"
plt.plot(epochs, original_val_loss, 'b+', label='Original model')
# "bo" is for "blue dot"
plt.plot(epochs, smaller_model_val_loss, 'bo', label='Smaller model')
plt.xlabel('Epochs')
plt.ylabel('Validation loss')
plt.legend()

plt.show()

```

```
Epoch 3/20
25000/25000 [==============================] - 2s 77us/step - loss: 0.2619 - acc
: 0.9214 - val_loss: 0.3364 - val_acc: 0.8842
Epoch 4/20
25000/25000 [==============================] - 3s 112us/step - loss: 0.2407 - ac
c: 0.9307 - val_loss: 0.3681 - val_acc: 0.8719
Epoch 5/20
25000/25000 [==============================] - 3s 114us/step - loss: 0.2286 - ac
c: 0.9364 - val_loss: 0.3667 - val_acc: 0.8745
Epoch 6/20
25000/25000 [==============================] - 2s 88us/step - loss: 0.2212 - acc
: 0.9392 - val_loss: 0.3767 - val_acc: 0.8718
Epoch 7/20
22016/25000 [========================>....] - ETA: 0s - loss: 0.2164 - acc: 0.9
422
```

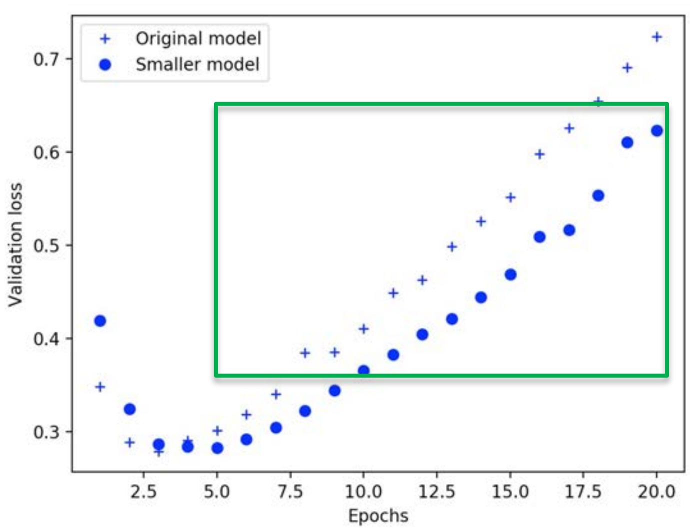


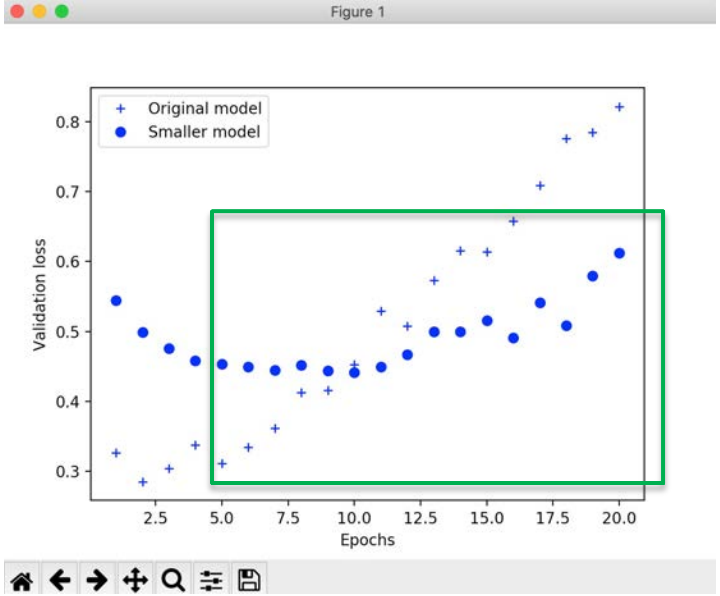
Practice 2

- Try other numbers

Listing 4.4 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'))
```





Method 2:
Adding weight regularization

- A Simple model
  - Less likely to overfit than complex ones
  - Distribution of parameter values
    - Less entropy (a model with fewer parameters)
- Put constraints
  - Forcing weights: small values
  - Make the distribution of weight values
    - More regular – weight regularization

Adding the loss function, Cost added

- L1 regularization: L1 norm, absolute value

of the weight coefficients

- L2 regularization: L2 norm, square value

Method 2: Adding weight regularization

- Add weight regularizer

```
# method 2: adding weight regularization
from tensorflow.keras import regularizers

l2_model = models.Sequential()
l2_model.add(layers.Dense(16, kernel_regularizer=regularizers.l2(0.001),
    activation='relu', input_shape=(10000,)))
l2_model.add(layers.Dense(16, kernel_regularizer=regularizers.l2(0.001),
    activation='relu'))
l2_model.add(layers.Dense(1, activation='sigmoid'))

l2_model.compile(optimizer='rmsprop', loss='binary_crossentropy', metrics=['acc'])
l2_model_hist = l2_model.fit(x_train, y_train, epochs=20, batch_size=512,
    validation_data=(x_test, y_test))

l2_model_val_loss = l2_model_hist.history['val_loss']
```

Import regularizers

Step1: add layers,
With regularizers, l2(0.001

Step 2: Compile
Setup the
configuration

Step 3:
Fit the model

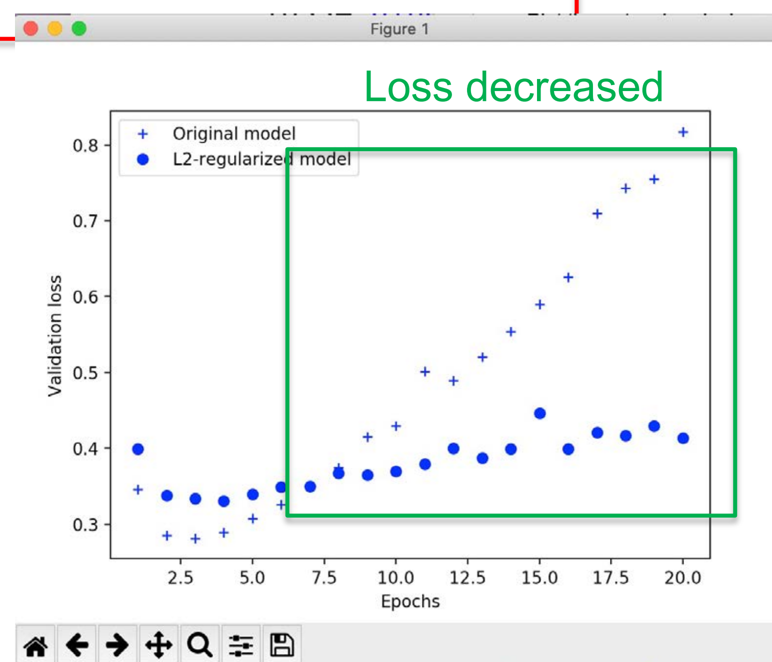
Assigning the loss results

Plot the regularization results

- Plot the results

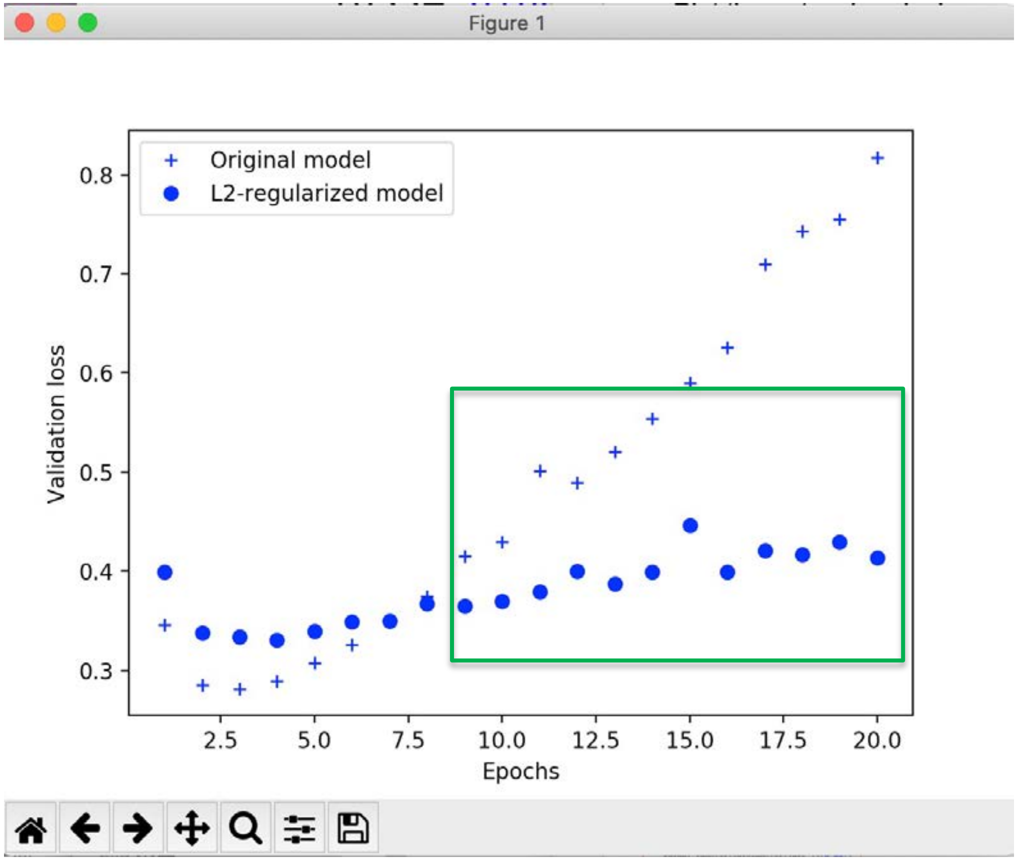
```
plt.plot(epochs, original_val_loss, 'b+', label='Original model')
plt.plot(epochs, l2_model_val_loss, 'bo', label='L2-regularized model')
plt.xlabel('Epochs')
plt.ylabel('Validation loss')
plt.legend()

plt.show()
```

```
Epoch 3/20
25000/25000 [==============================] - 2s 77us/step - loss: 0.2619 - acc
: 0.9214 - val_loss: 0.3364 - val_acc: 0.8842
Epoch 4/20
25000/25000 [==============================] - 3s 112us/step - loss: 0.2407 - ac
c: 0.9307 - val_loss: 0.3681 - val_acc: 0.8719
Epoch 5/20
25000/25000 [==============================] - 3s 114us/step - loss: 0.2286 - ac
c: 0.9364 - val_loss: 0.3667 - val_acc: 0.8745
Epoch 6/20
25000/25000 [==============================] - 2s 88us/step - loss: 0.2212 - acc
: 0.9392 - val_loss: 0.3767 - val_acc: 0.8718
Epoch 7/20
22016/25000 [========================>.....] - ETA: 0s - loss: 0.2164 - acc: 0.9
422
```



Practice 3

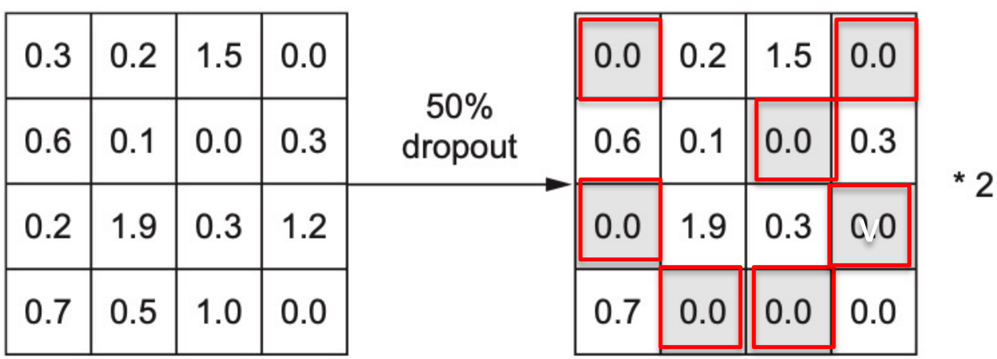
- Plot the network reducing results



Method 3:
Adding dropout

- Dropout
  - One of the most effective way,
    - commonly used
  - Regularization techniques
  - [Geoff Hinton et al, 14], U of Toronto
  - Applied to a layer (setting to zero)

Neural network matrix



Strange and arbitrary,
But really work!

Introduce some noise

Figure 4.8 Dropout applied to an activation matrix at training time, with rescaling happening during training. At test time, the activation matrix is unchanged.

Method 3: adding dropout

- Dropout

Step1: add layers,
With dropout

```
#method 3: adding dropout
dpt_model = models.Sequential()
dpt_model.add(layers.Dense(16, activation='relu', input_shape=(10000,)))
dpt_model.add(layers.Dropout(0.5))
dpt_model.add(layers.Dense(16, activation='relu'))
dpt_model.add(layers.Dropout(0.5))
dpt_model.add(layers.Dense(1, activation='sigmoid'))

dpt_model.compile(optimizer='rmsprop', loss='binary_crossentropy', metrics=['acc'])

dpt_model_hist = dpt_model.fit(x_train, y_train, epochs=20, batch_size=512,
    validation_data=(x_test, y_test))

dpt_model_val_loss = dpt_model_hist.history['val_loss']
```

Step 2: Compile, Setup the configuration

Step 3:
Fit the model

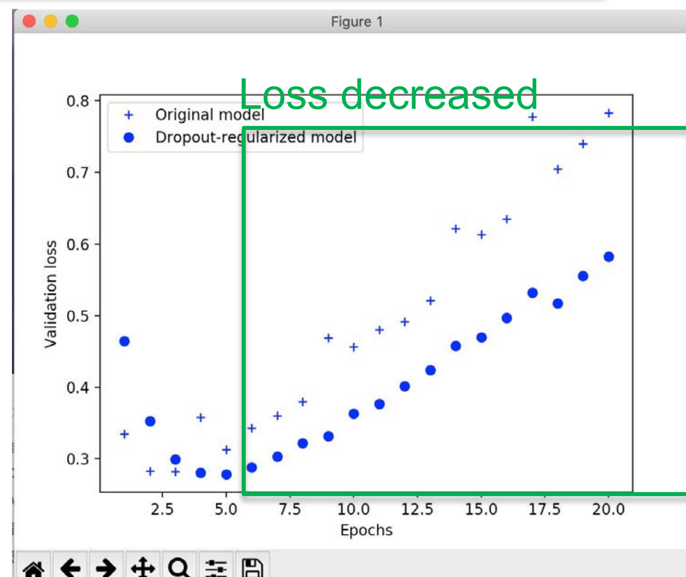
Assigning the loss results

Plot the network reducing results

- Plot the results

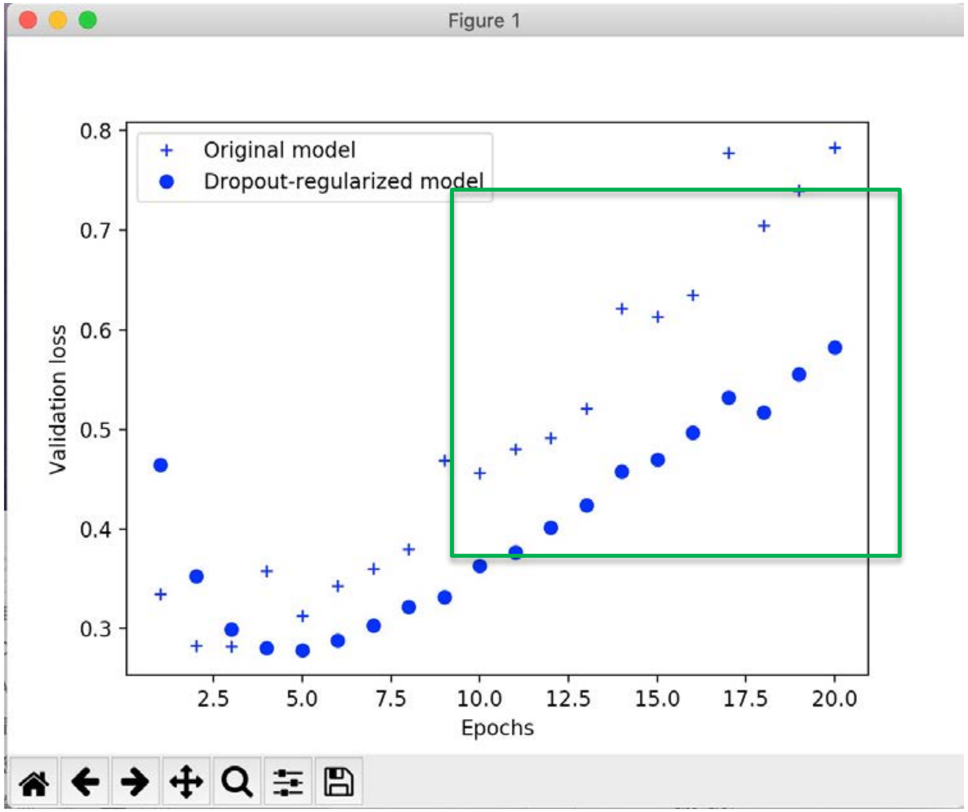
```
plt.plot(epochs, original_val_loss, 'b+', label='Original model')
plt.plot(epochs, dpt_model_val_loss, 'bo', label='Dropout-regularized model')
plt.xlabel('Epochs')
plt.ylabel('Validation loss')
plt.legend()

plt.show()
```

```
Epoch 3/20
25000/25000 [==============================] - 2s 77us/step - loss: 0.2619 - acc
: 0.9214 - val_loss: 0.3364 - val_acc: 0.8842
Epoch 4/20
25000/25000 [==============================] - 3s 112us/step - loss: 0.2407 - ac
c: 0.9307 - val_loss: 0.3681 - val_acc: 0.8719
Epoch 5/20
25000/25000 [==============================] - 3s 114us/step - loss: 0.2286 - ac
c: 0.9364 - val_loss: 0.3667 - val_acc: 0.8745
Epoch 6/20
25000/25000 [==============================] - 2s 88us/step - loss: 0.2212 - acc
: 0.9392 - val_loss: 0.3767 - val_acc: 0.8718
Epoch 7/20
22016/25000 [========================>....] - ETA: 0s - loss: 0.2164 - acc: 0.9
422
```



Practice 4

- Plot the dropout results



Full code (1/5)

```
from tensorflow.keras.datasets import imdb
import numpy as np

(train_data, train_labels), (test_data, test_labels) = imdb.load_data(num_words=10000)

def vectorize_sequences(sequences, dimension=10000):
    # Create an all-zero matrix of shape (len(sequences), dimension)
    results = np.zeros((len(sequences), dimension))
    for i, sequence in enumerate(sequences):
        results[i, sequence] = 1.  # set specific indices of results[i] to 1s
    return results

# Our vectorized training data
x_train = vectorize_sequences(train_data)
# Our vectorized test data
x_test = vectorize_sequences(test_data)
# Our vectorized labels
y_train = np.asarray(train_labels).astype('float32')
y_test = np.asarray(test_labels).astype('float32')

from tensorflow.keras import models
from tensorflow.keras import layers

```

Full code (2/5)

```
#reducing the network's size
original_model = models.Sequential()
original_model.add(layers.Dense(16, activation='relu', input_shape=(10000,)))
original_model.add(layers.Dense(16, activation='relu'))
original_model.add(layers.Dense(1, activation='sigmoid'))
original_model.compile(optimizer='rmsprop', loss='binary_crossentropy', metrics=['acc'])

original_hist = original_model.fit(x_train, y_train, epochs=20, batch_size=512,
    validation_data=(x_test, y_test))

epochs = range(1, 21)
original_val_loss = original_hist.history['val_loss']

original_val_acc = original_hist.history['val_acc']

import matplotlib.pyplot as plt

# b+ is for "blue cross"
plt.plot(epochs, original_val_loss, 'b+', label='Original model - loss')

plt.plot(epochs, original_val_acc, 'bo', label='Original model - acc')

plt.xlabel('Epochs')
plt.ylabel('Validation loss')
plt.legend()
plt.show()
```

Full code (3/5)

```
#method 1: reducing the network's size
smaller_model = models.Sequential()
smaller_model.add(layers.Dense(4, activation='relu', input_shape=(10000,)))
smaller_model.add(layers.Dense(4, activation='relu'))
smaller_model.add(layers.Dense(1, activation='sigmoid'))
smaller_model.compile(optimizer='rmsprop', loss='binary_crossentropy', metrics=['acc'])

smaller_model_hist = smaller_model.fit(x_train, y_train, epochs=20, batch_size=512,
    validation_data=(x_test, y_test))

smaller_model_val_loss = smaller_model_hist.history['val_loss']

# b+ is for "blue cross"
plt.plot(epochs, original_val_loss, 'b+', label='Original model')
# "bo" is for "blue dot"
plt.plot(epochs, smaller_model_val_loss, 'bo', label='Smaller model')
plt.xlabel('Epochs')
plt.ylabel('Validation loss')
plt.legend()

plt.show()
```

Full code (4/5)

```
# method 2: adding weight regularization
from tensorflow.keras import regularizers

l2_model = models.Sequential()
l2_model.add(layers.Dense(16, kernel_regularizer=regularizers.l2(0.001),
    activation='relu', input_shape=(10000,)))
l2_model.add(layers.Dense(16, kernel_regularizer=regularizers.l2(0.001),
    activation='relu'))
l2_model.add(layers.Dense(1, activation='sigmoid'))

l2_model.compile(optimizer='rmsprop', loss='binary_crossentropy', metrics=['acc'])
l2_model_hist = l2_model.fit(x_train, y_train, epochs=20, batch_size=512,
    validation_data=(x_test, y_test))

l2_model_val_loss = l2_model_hist.history['val_loss']

plt.plot(epochs, original_val_loss, 'b+', label='Original model')
plt.plot(epochs, l2_model_val_loss, 'bo', label='L2-regularized model')
plt.xlabel('Epochs')
plt.ylabel('Validation loss')
plt.legend()

plt.show()
```

Full code (5/5)

```
#method 3: adding dropout
dpt_model = models.Sequential()
dpt_model.add(layers.Dense(16, activation='relu', input_shape=(10000,)))
dpt_model.add(layers.Dropout(0.5))
dpt_model.add(layers.Dense(16, activation='relu'))
dpt_model.add(layers.Dropout(0.5))
dpt_model.add(layers.Dense(1, activation='sigmoid'))

dpt_model.compile(optimizer='rmsprop', loss='binary_crossentropy', metrics=['acc'])

dpt_model_hist = dpt_model.fit(x_train, y_train, epochs=20, batch_size=512,
    validation_data=(x_test, y_test))

dpt_model_val_loss = dpt_model_hist.history['val_loss']

plt.plot(epochs, original_val_loss, 'b+', label='Original model')
plt.plot(epochs, dpt_model_val_loss, 'bo', label='Dropout-regularized model')
plt.xlabel('Epochs')
plt.ylabel('Validation loss')
plt.legend()

plt.show()
```

Summary

- 3 methods to avoid overfitting:
  - Reduce the network's size
  - Add weight regularization
  - Add dropout

The universal workflow of machine learning (1/4)

- Defining the problem and assembling a dataset
  - What will your input data be?
  - What are you trying to predict?
  - What type of problem are you facing?
    - Binary classification
    - Multiclass classification
    - Scalar regression
    - Multiclass, multilabel classification
  - Input/output: I/O should be defined
    - What data you' ll have?

The universal workflow of machine learning (2/4)

- Choosing a measure of success
  - Accuracy
  - Precision/recall
  - ROC: receiver operating characteristic curve
- Deciding on an evaluation protocol
  - Hold-out validation: having plenty of data
  - K-fold cross-validation: too few samples
  - Iterated K-fold validation: highly accurate model, with little data

In most cases, the first (hold-out validation) will work well enough.

The universal workflow of machine learning (3/4)

- Preparing your data
  - The data should be formatted as tensors
  - Values to be scaled to small values
    - [-1,1] or [0,1]
  - Normalizing for different features
  - Feature engineering
- Developing a model that does better than baseline
  - MNIST digits 0~9: baseline accuracy: 0.1; 10%
    - Should be better than random guess

The universal workflow of machine learning (4/4)

- Last-layer activation, loss function, optimization configuration

Table 4.1 Choosing the right last-layer activation and loss function for your model

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

- Scaling up: developing a model that overfits
  - Add layers, make the layers bigger, train more epochs
- Regularization: add dropout, add L2 regularization

Textbook Reading

- *Deep Learning with Python*
  - Ch 4, Fundamentals of machine learning
    - p.94 - p.116