2021/12/01

Validation Curve and Learning Curve

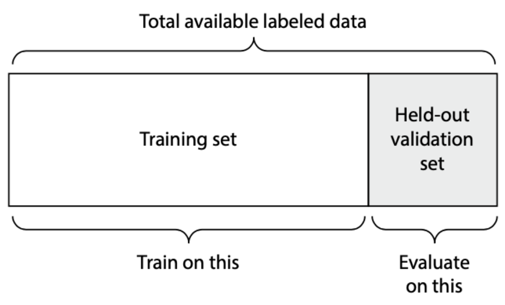
Validation curve and learning curve (or sometimes short for learning curves) are important topics for machine learning (and deep learning). The validation curve summarizes **the tradeoff between training and validation errors as we vary the model complexity**. The learning curve summarizes **the tradeoff between training and validation errors as we vary the size of the training set**.

Food for Thought

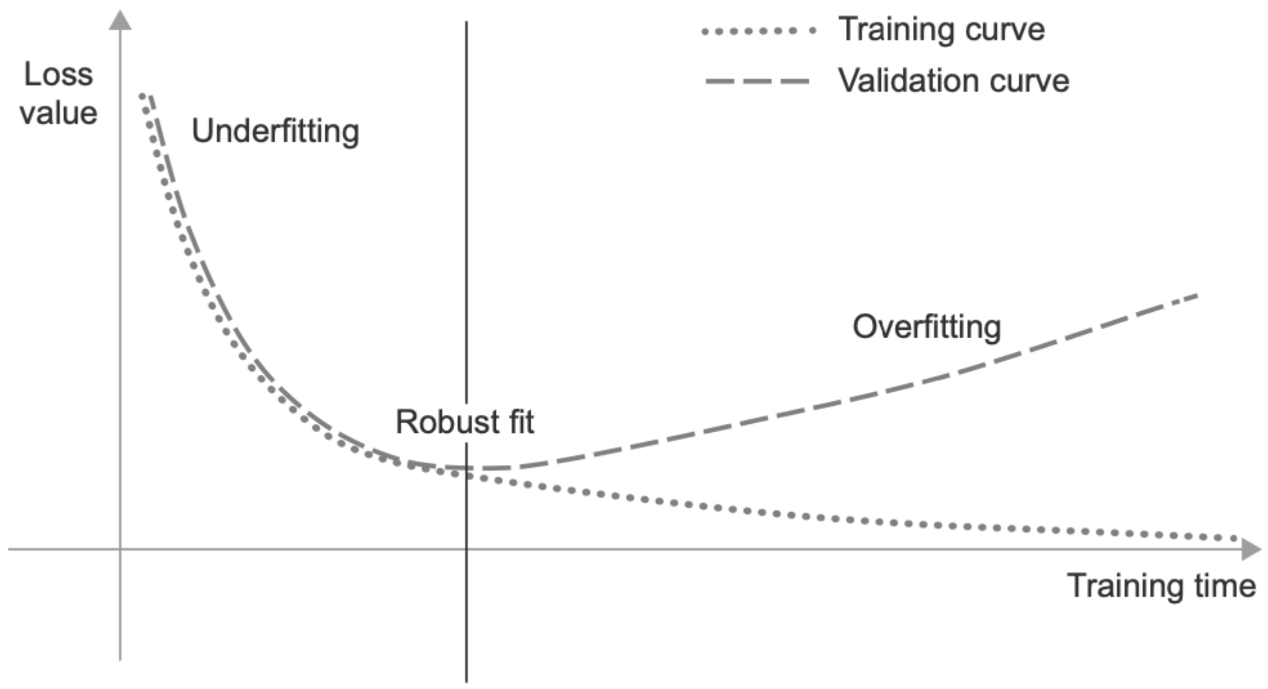
1. The fundamental issue in machine learning (and deep learning) is the tension between **optimization and generalization**. Optimization refers to the process of adjusting a model to get the best performance possible on the training data (the learning in machine learning), whereas generalization refers to how well the trained model performs on data it has never seen before.
2. The goal of the game is to get good generalization, of course, but you don't control generalization; you can only fit the model to its training data. If you do that too well, overfitting kicks in and generalization suffers.
3. In classical machine learning, we use the **cross validation** technique to help us identify overfit and pick up the model that perform best on generalization.
4. In deep learning, we do not have the computational luxury to do cross validation and often rely on the **validation curve** to help us identify overfit and pick up the model that perform best on generalization.

1. Validation Curve

In deep learning, we often do a simple holdout validation and split the data into training set and validation set.



When we increase the number of **epochs** (model complexity) in deep learning, we would expect the loss from training curve and validation curve to behave like:



1. At the beginning of training, optimization and generalization are correlated: the lower the loss on training data, the lower the loss on test data. While this is happening, your 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.
2. After a certain number of iterations on the training data, generalization stops improving, 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.

Fun Time: We should use a deep learning model in (1) underfit region (2) robust fit region (3) overfit region.

2. Example

In the following, we will first use IMDB dataset: a set of 50,000 highly polarized reviews from the Internet Movie Database to illustrate the overfitting problem for neural network.

2.1 The IMDB dataset

You'll work with the IMDB dataset: a set of 50,000 highly polarized reviews from the Internet Movie

Database. They're split into 25,000 reviews for training and 25,000 reviews for testing, each set consisting of 50% negative and 50% positive reviews.

Just like the MNIST dataset, the IMDB dataset comes packaged with `Keras`. It has already been preprocessed: the reviews (sequences of words) have been turned into sequences of integers, where each integer stands for a specific word in a dictionary.

The following code will load the dataset (when you run it the first time, about 80 MB of data will be downloaded to your machine).

```
from tensorflow.keras.datasets import imdb

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

The argument `num_words=10000` means that we will only keep the top 10,000 most frequently occurring words in the training data. Rare words will be discarded. This allows us to work with vector data of manageable size.

The variables `train_data` and `test_data` are lists of reviews, each review being a list of word indices (encoding a sequence of words). `train_labels` and `test_labels` are lists of 0s and 1s, where 0 stands for "negative" and 1 stands for "positive":

For example, if we type

```
train_data[0]
```

We see a list of word indices encoding a sequence of words for the first review:

```
Out[2]: [1,
         14,
         22,
         16,
         43,
         530,
         973,
         1622,
         1385,
         65,
         458,
         4468,
         66,
         3941,
         4,
         173,
         36,
         256,
```

…

If we type

```
train_labels[0]
```

We see the first review results:

```
Out[3]: 1
```

Fun Time: Is the first review positive or negative? (1) positive (2) negative.

You can quickly decode one of these reviews back to English words, for example, for the first review:

```
# word_index is a dictionary mapping words to an integer index
word_index = imdb.get_word_index()
# We reverse it, mapping integer indices to words
reverse_word_index = dict([(value, key) for (key, value) in
word_index.items()])
# We decode the review; note that our indices were offset by 3
# because 0, 1 and 2 are reserved indices for "padding",
# "start of sequence", and "unknown".
decoded_review = ' '.join([reverse_word_index.get(i - 3, '?') for i in
train_data[0]])

decoded_review
```

```
Out[5]: "? this film was just brilliant casting location scenery story direction everyone's really suited the part they playe
        d and you could just imagine being there robert ? is an amazing actor and now the same being director ? father came f
        rom the same scottish island as myself so i loved the fact there was a real connection with this film the witty remar
        ks throughout the film were great it was just brilliant so much that i bought the film as soon as it was released for
        ? and would recommend it to everyone to watch and the fly fishing was amazing really cried at the end it was so sad a
        nd you know what they say if you cry at a film it must have been good and this definitely was also ? to the two littl
        e boy's that played the ? of norman and paul they were just brilliant children are often left out of the ? list i thi
        nk because the stars that play them all grown up are such a big profile for the whole film but these children are ama
        zing and should be praised for what they have done don't you think the whole story was so lovely because it was true
        and was someone's life after all that was shared with us all"
```

2.2 Preparing the data

You can't feed lists of integers into a neural network. You have to turn your lists into tensors. One way to do it is through on-hot encoding. One-hot encode your lists to turn them into vectors of 0s and 1s. This would mean, for instance, turning the sequence [3, 5] into a 10,000-dimensional vector that would be all 0s except for indices 3 and 5, which would be 1s. Then you could use as the first layer in your network a Dense layer, capable of handling floating-point vector data.

Let's vectorize our data, which we will do manually for maximum clarity:

```
import numpy as np

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)
```

Here's what the first sample look like now:

```
x_train[0]
```

```
Out[9]: array([0., 1., 1., ..., 0., 0., 0.])
```

You should also vectorize your labels, which is straightforward:

```
# Our vectorized labels
y_train = np.asarray(train_labels).astype('float32')
y_test = np.asarray(test_labels).astype('float32')
```

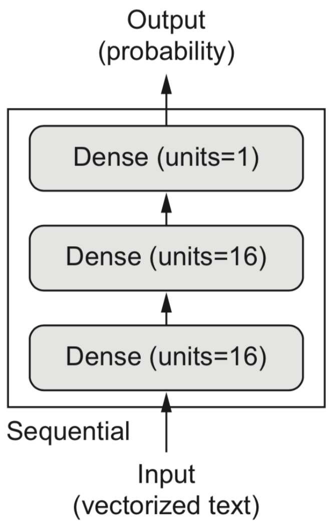
Now the data is ready to be fed into a neural network.

2.3 Building your network

For this example, the input data is vectors, and the labels are scalars (1s and 0s): this is the easiest setup you'll ever encounter. A type of network that performs well on such a problem is a simple stack of fully connected (Dense) hidden layers with RELU activations: `Dense(16, activation='relu')`.

For this example, we will make the following architecture choice:
1. Two intermediate layers with 16 hidden units each
2. A third layer that will output the scalar prediction regarding the sentiment of the current review

Figure below shows what the network looks like.



And here's the `Keras` implementation, similar to the MNIST example you saw previously.

```
from tensorflow.keras import models
from tensorflow.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'))
```

Q: How many training parameters for this network model?
A:
$$(10000 + 1) * 16 + (16 + 1) * 16 + (16 + 1) * 1 = 160305$$

Finally, you need to choose a loss function and an optimizer. Because you're facing a binary classification problem and the output of your network is a probability (you end your network with a single-unit layer with a sigmoid activation), it's best to use the `binary_crossentropy` loss. It isn't the only viable choice: you could use, for instance, `mean_squared_error`. But `crossentropy` is usually the best choice when you're dealing with models that output probabilities.

Here's the step where you configure the model with the `rmsprop` optimizer and the `binary_crossentropy` loss function. Note that you'll also monitor accuracy during training.

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

2.4 Validating your approach

In order to monitor during training the accuracy of the model on data it has never seen before, you'll create a validation set by setting apart 10,000 samples from the original training data.

```
x_val = x_train[:10000]
partial_x_train = x_train[10000:]

y_val = y_train[:10000]
partial_y_train = y_train[10000:]
```

We will now train our model for 20 epochs (20 iterations over all samples in the `partial_x_train` and `partial_y_train` tensors), in mini-batches of 512 samples. At this same time we will monitor loss and accuracy on the 10,000 samples that we set apart. This is done by passing the validation data as the `validation_data` argument:

```
history = model.fit(partial_x_train,
                    partial_y_train,
                    epochs=20,
                    batch_size=512,
                    validation_data=(x_val, y_val))
```

```
Epoch 1/20
30/30 [==============================] - 2s 59ms/step - loss: 0.5150 - accuracy: 0.7817 - val_loss: 0.3923 - val_accuracy: 0.8704
Epoch 2/20
30/30 [==============================] - 1s 41ms/step - loss: 0.3129 - accuracy: 0.9027 - val_loss: 0.3233 - val_accuracy: 0.8740
Epoch 3/20
30/30 [==============================] - 1s 41ms/step - loss: 0.2314 - accuracy: 0.9244 - val_loss: 0.2817 - val_accuracy: 0.8910
Epoch 4/20
30/30 [==============================] - 1s 42ms/step - loss: 0.1839 - accuracy: 0.9414 - val_loss: 0.2834 - val_accuracy: 0.8863
Epoch 5/20
30/30 [==============================] - 1s 43ms/step - loss: 0.1511 - accuracy: 0.9511 - val_loss: 0.2755 - val_accuracy: 0.8914
```

…

Note that the call to `model.fit()` returns a `History` object. This object has a member `history`, which is a dictionary containing data about everything that happened during training. Let's look at it:

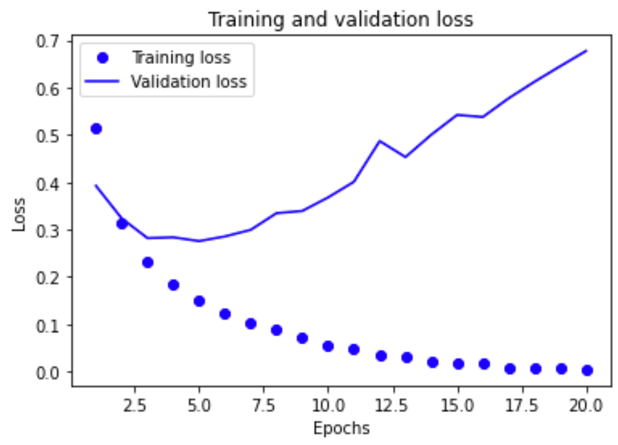
```
history_dict = history.history
history_dict.keys()
```

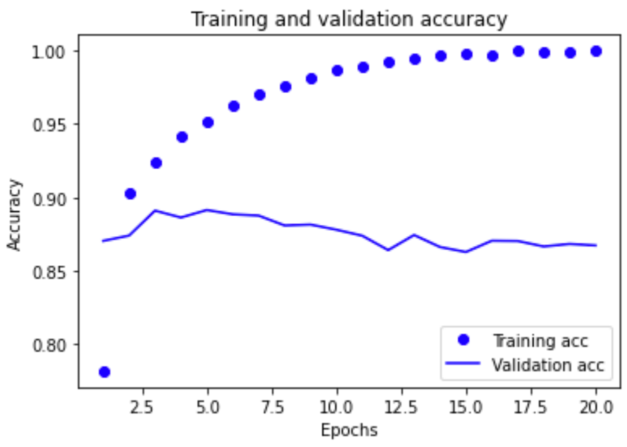
```
dict_keys(['loss', 'accuracy', 'val_loss', 'val_accuracy'])
```

It contains 4 entries: one per metric that was being monitored, during training and during validation. Let's use Matplotlib to plot the training and validation loss side by side, as well as the training and validation accuracy:

```
import matplotlib.pyplot as plt

acc = history.history['accuracy']
val_acc = history.history['val_accuracy']
loss = history.history['loss']
val_loss = history.history['val_loss']

epochs = range(1, len(acc) + 1)

# "bo" is for "blue dot"
plt.plot(epochs, loss, 'bo', label='Training loss')
# b is for "solid blue line"
plt.plot(epochs, val_loss, 'b', label='Validation loss')
plt.title('Training and validation loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()

plt.show()
```

```
plt.clf()   # clear figure
acc_values = history_dict['accuracy']
val_acc_values = history_dict['val_accuracy']

plt.plot(epochs, acc, 'bo', label='Training acc')
plt.plot(epochs, val_acc, 'b', label='Validation acc')
plt.title('Training and validation accuracy')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend()

plt.show()
```





Q: what have you observed from these plots?

A:

The training loss decreases with every epoch, and the training accuracy increases with every epoch. That's what you would expect when running gradient-descent optimization—the quantity you're trying to minimize should be less with every iteration. But that isn't the case for the validation loss and accuracy: they seem to peak at the fourth epoch. This is an example of what we warned against earlier: a model that performs better on the training data isn't necessarily a model that will do better on data it has never seen before. In precise terms, what you're seeing is overfitting: after the second epoch, you're overoptimizing on the training data, and you end up learning representations that are specific to the training data and don't generalize to data outside of the training set.

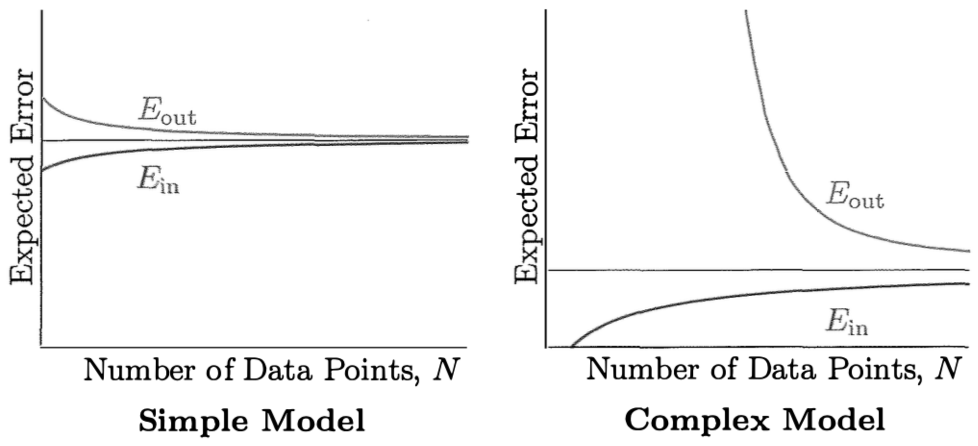
You can download the above code `Keras_IMDB.ipynb` from course website.

3. Learning Curve

We close the discussion with learning curve. One important aspect of model complexity is that **the optimal model will generally depend on the size of your training data**.

It is often useful to explore the behavior of the model as a function of the number of training points, which we can do by using increasingly larger subsets of the data to fit our model. A plot of the training/validation score with respect to the size of the training set is known as a **learning curve**.

We illustrate the learning curves for a simple learning model and a complex one:



Fun Time: For the simple model, what should you do if you would like to improve the performance of your deep learning model? (1) increase model complexity (2) label more training data

Fun Time: For the complex model, what should you do if you would like to improve the performance of your deep learning model? (1) increase model complexity (2) label more training data