Lecture_5_Fundamentals_of_ML

November 3, 2022

1 Fundamentals of machine learning

[1]:
```
import matplotlib.pyplot as plt

def plot_helper(x_data, y_data, plot_styles, labels, title, x_label, y_label):
    assert len(x_data) == len(y_data)
    assert len(x_data) == len(plot_styles)
    assert len(x_data) == len(labels)

    fig = plt.figure(figsize=(9 / 2.54, 5 / 2.54), dpi=150)
    plt.rc('axes', titlesize=7)  # fontsize of the axes title
    plt.rc('axes', labelsize=6)  # fontsize of the x and y labels
    plt.rc('xtick', labelsize=6)  # fontsize of the tick labels
    plt.rc('ytick', labelsize=6)  # fontsize of the tick labels
    plt.autoscale(enable=True, axis='x', tight=True)

    for i in range(len(x_data)):
        plt.plot(x_data[i], y_data[i], plot_styles[i], label=labels[i],␣
↪linewidth=1, markersize=3)
    plt.title(title)
    plt.xlabel(x_label)
    plt.ylabel(y_label)
    plt.legend(fontsize=5.5)
```

1.1 Generalization: The goal of machine learning

In the two examples presented in Lecture 4, namely predicting movie reviews and house-price regression, we split the data into a **training set**, a **validation set**, and a **test set**. The reason not to evaluate the models on the same data they were trained on is evident: after just a few epochs, performance on never-before-seen data started diverging from performance on the training data, which always improves as training progresses. The models started to **overfit**. Overfitting occurs in every ML problem.

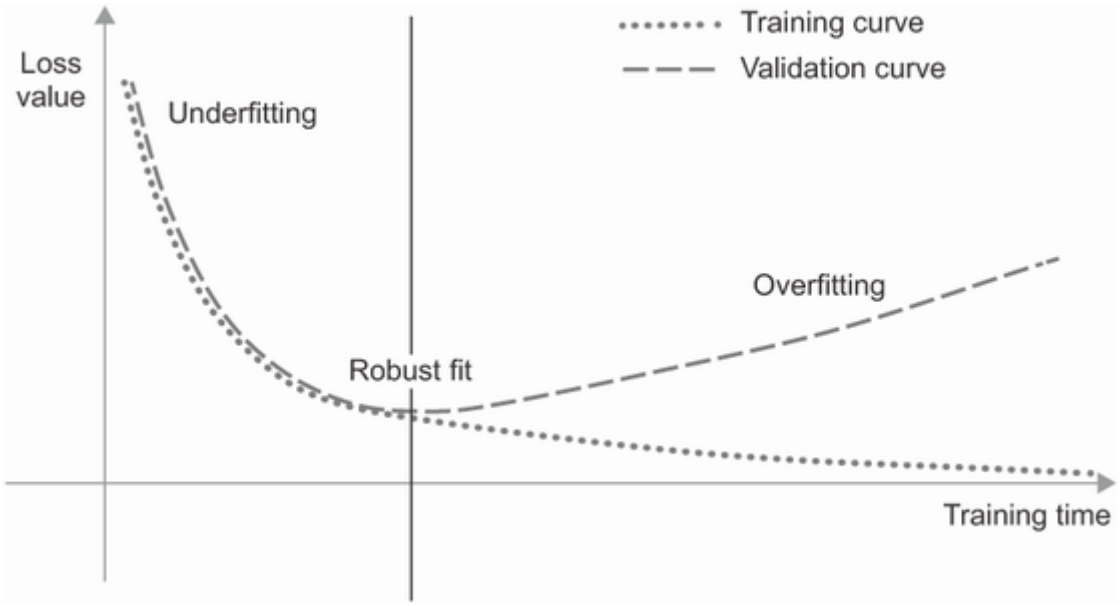
The fundamental issue in ML is **compromise between optimization and generalization**. **Optimization** refers to the process of adjusting a model to get the best performance possible on the training data (the "learning" in the context of ML), whereas **generalization** refers to how well the trained model performs on data it has never seen before. The objective is to get good generalization, but we don't control generalization! We can only fit the model to its training data.

If we do that too well, overfitting takes place and generalization suffers.

What causes overfitting and how to we achieve good generalization?

1.1.1 Underfitting and overfitting

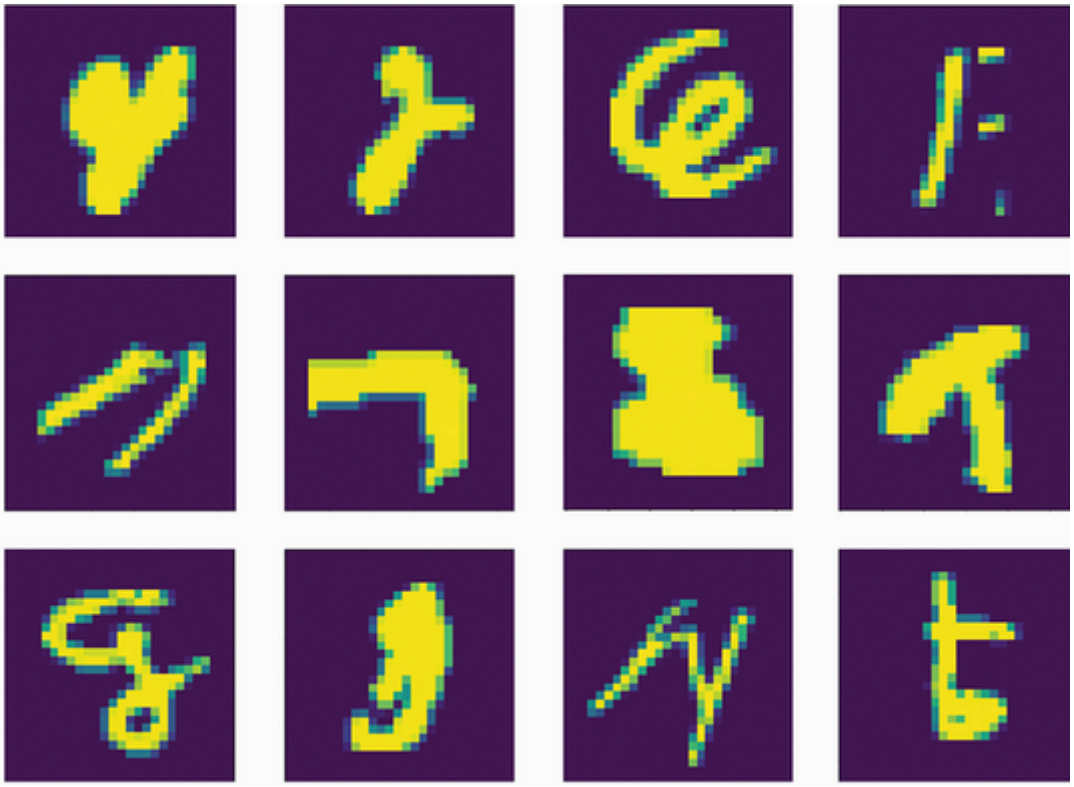
For the models we used in Lecture 4, performance on the validation data started improving as training went on and then inevitably started degrading. This pattern (illustrated below) is universal.



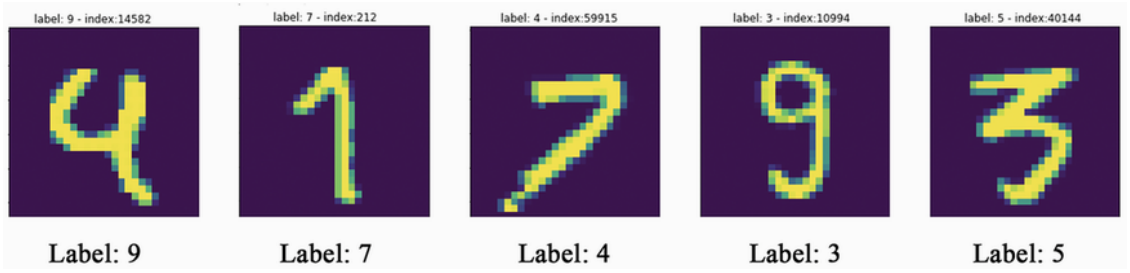
At the beginning of training, optimization and generalization are correlated: the lower the loss on training data, the lower the loss on test data. While this is happening, our model is said to be **underfit**: there is still progress to be made. The network hasn't learnt all relevant patterns in the training data yet. But 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's beginning of *learning patterns specific to the training data*, but are misleading or irrelevant with new data.

Overfitting is particularly likely to occur when your data is noisy, if it involves uncertainty, or if it includes rare features. Let's look at concrete examples.

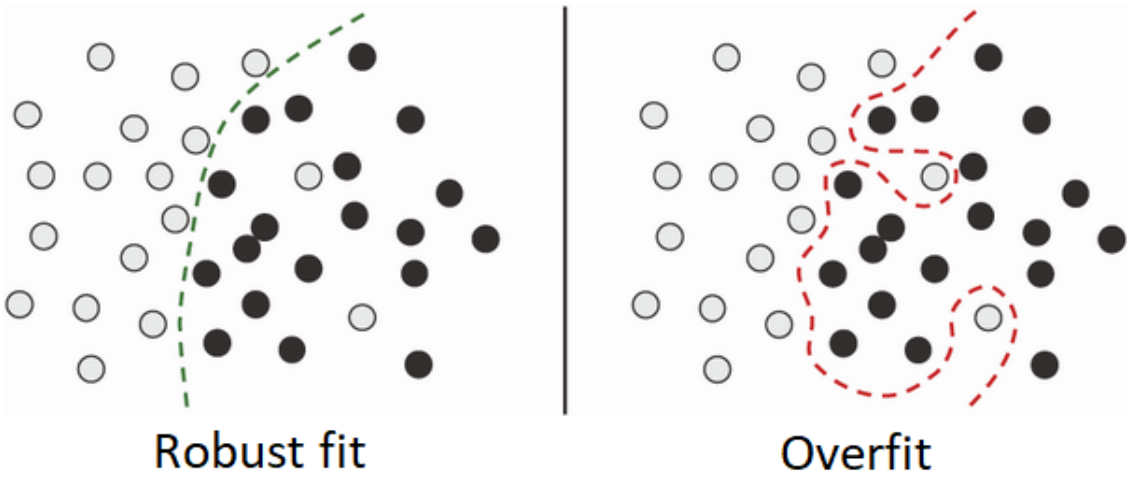
Noisy training data In real-world datasets, it's very common for some inputs to be invalid. For example, look at the MNIST images given below. Can you say, with confidence, what digits they correspond to?



Even worse, we can have perfectly valid inputs which end up mislabeled:

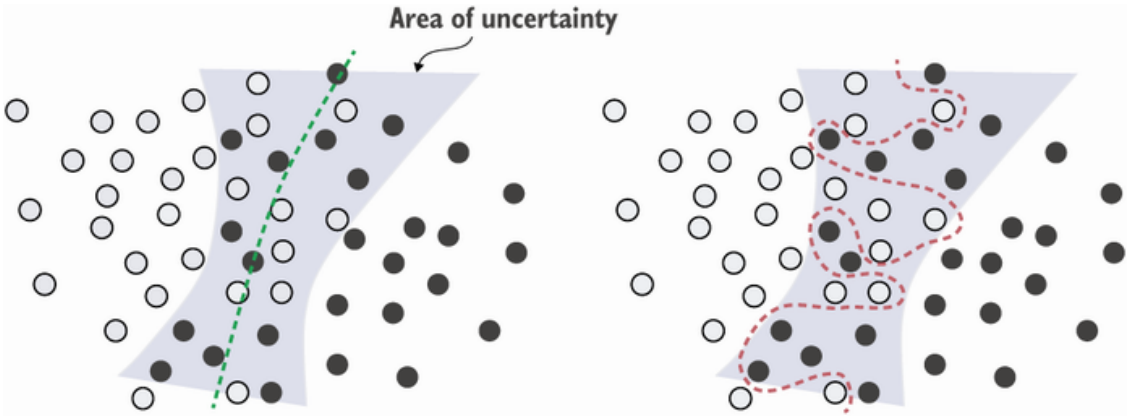


Trying to incorporate these **outliers** will lead to degradation of generalization performance, as shown below.



Ambiguous features Not all data noise comes from inaccuracies — even perfectly clean and neatly labeled data can be noisy when the problem involves uncertainty and ambiguity. *In classification tasks, it is often the case that some regions of the input feature space are associated with multiple classes at the same time.* Let's say we're developing a model that takes an image of a banana and predicts whether the banana is unripe, ripe, or rotten. These categories have no objective boundaries, so the same picture might be classified as either unripe or ripe by different human labelers. Similarly, many problems involve randomness. You could use atmospheric pressure data to predict whether it will rain tomorrow, but the exact same measurements may be followed sometimes by rain and sometimes by a clear sky, with some probability.

A model could overfit by being too confident about ambiguous regions of the feature space (see figure below (right)). A more robust fit would ignore individual data points and look at the bigger picture (figure left).



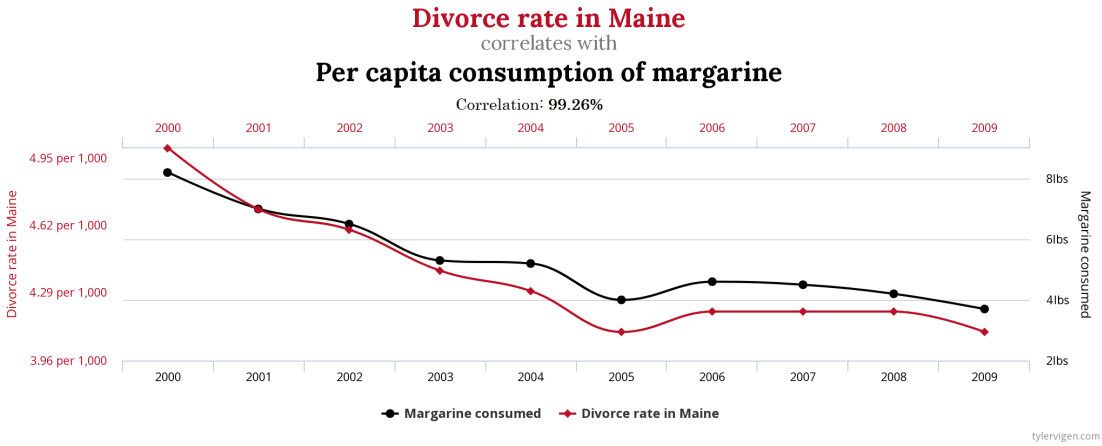
Rare features and spurious correlations An example from life. If you've been bitten twice in your life, both times by a yellow dog, you might infer that all yellow dogs are dangerous and you should keep away from them. That's **overfitting** due to a **spurious (false) correlation**. The color of a dog is not correlated with its character.

ML models trained on datasets that include rare feature values are highly susceptible to overfitting. In a sentiment classification task, if the word "cherimoya" (a fruit native to the Andes) only appears

in one text in the training data, and this text happens to be negative in sentiment, a poorly regularized model might put a very high weight on this word and always classify new texts that mention cherimoyas as negative.

A feature value that occurs multiple times can also lead to spurious correlations. Consider a word that occurs in 100 samples in your training data and that's associated with a positive sentiment 54% of the time and with a negative sentiment 46% of the time. That difference may well be a complete statistical fluke, yet your model is likely to learn to leverage that feature for its classification task. This is one of the most common sources of overfitting.

A spurious correlation also occurs when two variables are correlated, but don't have a causal relationship. In other words, it appears like values of one variable cause changes in the other variable, but that's not actually happening. Consider, for example, the correlation between divorce rate in Maine and per capita consumption of margarine in period 2000-2009.



Here's a striking example! Take MNIST. Create a new training set by concatenating 784 noise dimensions to the existing 784 (28×28) dimensions, so half of the data is noise. For comparison, create an additional equivalent dataset by concatenating 784 all-zeros dimensions. Our concatenation of meaningless features does not at all affect the information content of the data: we're only adding something. Human classification accuracy wouldn't be affected by these transformations at all.

Adding noise channels or all-zeros channels to MNIST

[2]:
```
from tensorflow.keras.datasets import mnist
import numpy as np

(train_images, train_labels), _ = mnist.load_data()
train_images = train_images.reshape((60000, 28 * 28))
train_images = train_images.astype("float32") / 255

train_images_with_noise_channels = np.concatenate(
    [train_images, np.random.random((len(train_images), 784))], axis=1)

train_images_with_zeros_channels = np.concatenate(
```

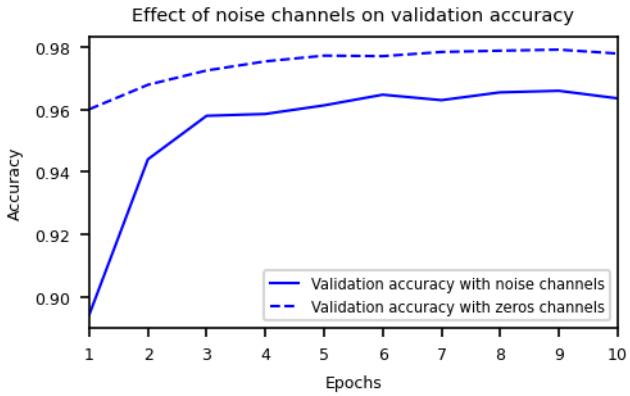
```
    [train_images, np.zeros((len(train_images), 784))], axis=1)
```

Training the same model on MNIST data with noise channels or all-zero channels

[3]:
```
from tensorflow import keras
from tensorflow.keras import layers

def get_model():
    model = keras.Sequential([
        layers.Dense(512, activation="relu"),
        layers.Dense(10, activation="softmax")
    ])
    model.compile(optimizer="rmsprop",
                  loss="sparse_categorical_crossentropy",
                  metrics=["accuracy"])
    return model

model = get_model()
history_noise = model.fit(
    train_images_with_noise_channels, train_labels,
    epochs=10,
    batch_size=128,
    validation_split=0.2,
    verbose=0)

model = get_model()
history_zeros = model.fit(
    train_images_with_zeros_channels, train_labels,
    epochs=10,
    batch_size=128,
    validation_split=0.2,
    verbose=0)
```

Plotting a validation accuracy comparison

[4]:
```
val_acc_noise = history_noise.history["val_accuracy"]
val_acc_zeros = history_zeros.history["val_accuracy"]
epochs = range(1, len(val_acc_noise) + 1)
plot_helper([epochs, epochs],
            [val_acc_noise, val_acc_zeros],
            ["b-", "b--"],
            ["Validation accuracy with noise channels", "Validation accuracy␣
 ↪with zeros channels"],
            "Effect of noise channels on validation accuracy",
            "Epochs",
            "Accuracy")
```



Despite the data holding the same information in both cases, the validation accuracy of the model trained with noise channels is about one percent lower — purely through the influence of spurious correlations. The more noise channels you add, the further accuracy will degrade.

Feature selection

Noisy features inevitably lead to overfitting. As such, in cases where you aren't sure whether the features you have are informative or distracting, it's common to do **feature selection** before training. Restricting the IMDB data to the top 10,000 most common words was a crude form of feature selection, for instance. The typical way to do feature selection is to compute some usefulness score for each feature available — a measure of how informative the feature is with respect to the task, such as the **mutual information between the feature and the labels** — and only keep features that are above some threshold. Doing this would filter out the noise channels in the preceding example.

Another approach for feature selection is an **ablation study**. Ablation study investigates the performance of an ML system by removing certain components to understand the contribution of the component to the overall system. If after removing a certain feature, the performance of the system does not change at all, or at negligible rate, that feature can be removed from the training process.

1.1.2 The nature of generalization in deep learning

A remarkable fact about DL models is that they **can be trained to fit anything**, as long as they have enough representational power.

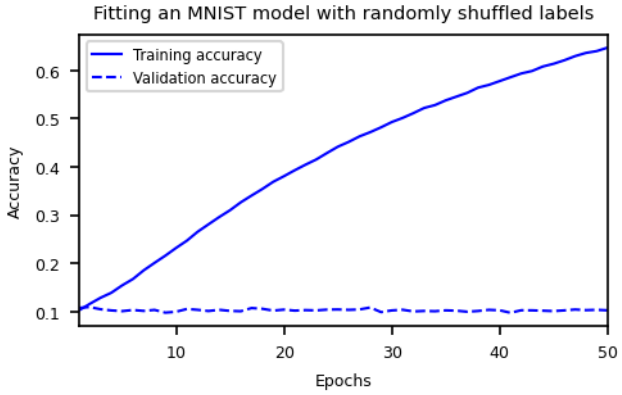
Let's shuffle the MNIST labels and train a model with the shuffled labels. Even though there is no relationship whatsoever between the inputs and the shuffled labels, the training loss decreases just fine, even with a relatively small model. On the other hand, the validation loss does not improve at all over time, since there is no possibility of generalization in this setting.

Fitting a MNIST model with randomly shuffled labels

[5]:
```
(train_images, train_labels), _ = mnist.load_data()
train_images = train_images.reshape((60000, 28 * 28))
```

```
train_images = train_images.astype("float32") / 255

random_train_labels = train_labels[:]
np.random.shuffle(random_train_labels)

model = keras.Sequential([
    layers.Dense(512, activation="relu"),
    layers.Dense(10, activation="softmax")
])
model.compile(optimizer="rmsprop",
              loss="sparse_categorical_crossentropy",
              metrics=["accuracy"])
history_random = model.fit(train_images, random_train_labels,
                           epochs=50,
                           batch_size=128,
                           validation_split=0.2,
                           verbose=0)
```

[6]:
```
acc_train = history_random.history["accuracy"]
acc_val = history_random.history["val_accuracy"]
epochs = range(1, len(acc_train) + 1)
plot_helper([epochs, epochs],
            [acc_train, acc_val],
            ["b-", "b--"],
            ["Training accuracy", "Validation accuracy"],
            "Fitting an MNIST model with randomly shuffled labels",
            "Epochs",
            "Accuracy")
```



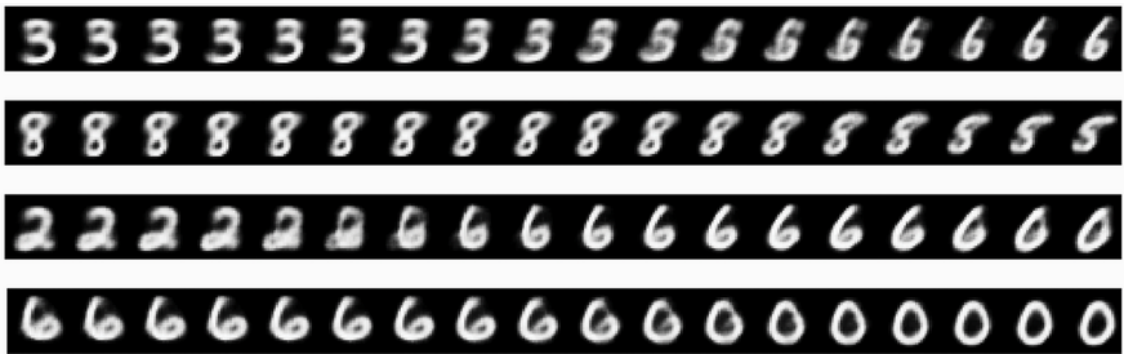
In fact, we don't even need to do this with MNIST data — we could just generate white noise inputs and random labels. You could fit a model on that, too, as long as it has enough parameters. It would just end up **memorizing specific inputs**.

If this is the case, then how come DL models generalize at all? Shouldn't they just learn an ad hoc mapping between training inputs and targets, like dictionaries does? What expectation can we have that this mapping will work for new inputs?

As it turns out, the nature of generalization in DL has rather little to do with DL models themselves, and much to do with the structure of information in the real world.

The manifold hypothesis The input to an MNIST classifier (before preprocessing) is a 28 × 28 array of integers between 0 and 255. The total number of possible input values is thus 256^784, much greater than the number of atoms in the observable universe (approx. 10^82 atoms). However, very few of these inputs would look like valid MNIST samples, i.e. actual handwritten digits only occupy a negligible subspace of the parent space of all possible 28 × 28 `uint8` arrays. What's more, this subspace isn't just a set of points sprinkled at random in the parent space: it is highly structured.

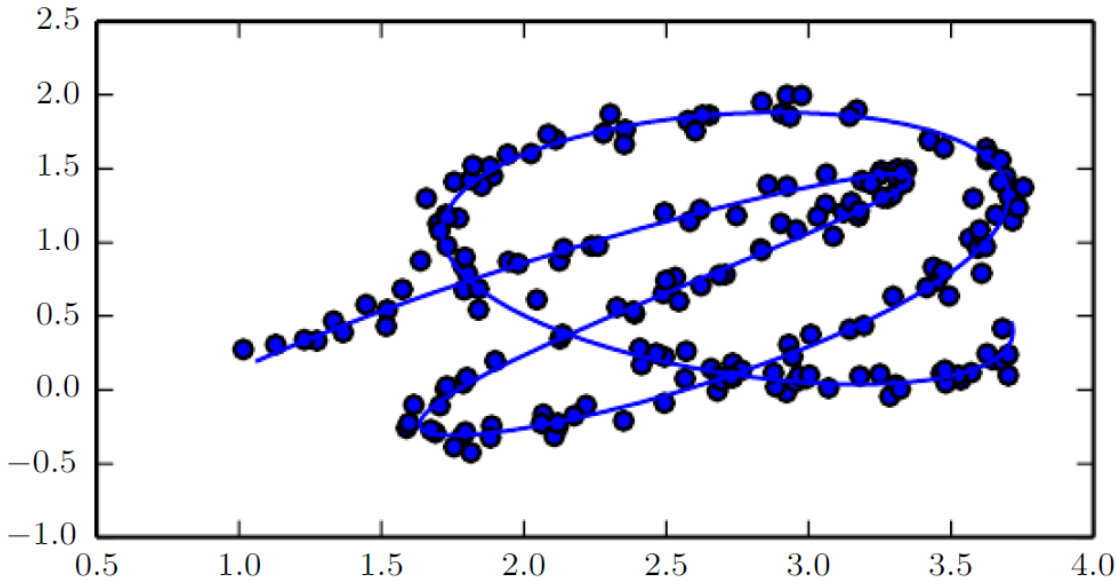
First, the **subspace of valid handwritten digits is continuous**: if you take a sample and modify it a little, it will still be recognizable as the same handwritten digit. Further, **all samples in the valid subspace are connected by smooth paths that run through the subspace**. This means that if you take two random MNIST digits A and B, there exists a sequence of "intermediate" images that morph A into B, such that two consecutive digits are very close to each other (in the figure below, different MNIST digits gradually morph into one another). Perhaps there will be a few ambiguous shapes close to the boundary between two classes, but even these shapes would still look very digit-like.



In technical terms, we would say that handwritten digits form a **manifold** within the space of possible 28 × 28 `uint8` arrays. **A "manifold" is a connected region, a lower-dimensional subspace of some parent space that is locally similar to a linear space**. For instance, a smooth curve in the plane is a 1D manifold within a 2D space, because for every point of the curve, you can draw a tangent (the curve can be approximated by a line at every point). A smooth surface within a 3D space is a 2D manifold. And so on.

Although there is a formal mathematical meaning to the term "manifold", in ML it tends to be used more loosely to designate a connected set of points that can be approximated well by considering only a small number of dimensions, embedded in a higher-dimensional space. Take, for example, training data lying near a 1D manifold embedded in 2D space (figure below). In the context of ML, we allow the dimensionality of the manifold to vary from one point to another. This often happens when a manifold intersects itself. For example, a digit 8 is a manifold that has a single dimension

in most places but two dimensions at the intersection at the center. Solid line in the figure below indicates the underlying manifold that our ML model should infer.



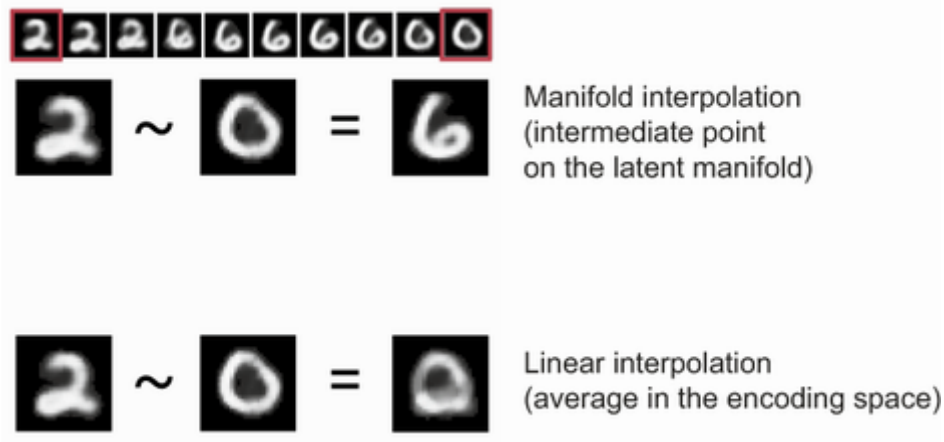
More generally, the manifold hypothesis states that all natural data lies on a low-dimensional manifold within the high-dimensional space where it is encoded. That's a pretty strong statement about the structure of information in the universe. As far as we know, it's accurate, and it's the reason why DL works. It's true for MNIST digits, but also for human faces, tree morphology, the sounds of the human voice, and even natural language.

The **manifold hypothesis** implies that: - ML models only have to fit relatively simple, low-dimensional, highly structured subspaces (latent manifolds) within their potential input space. - Within one of these manifolds, it's always possible to interpolate between two inputs, i.e. morph one into another via a continuous path along which all points fall on the manifold.

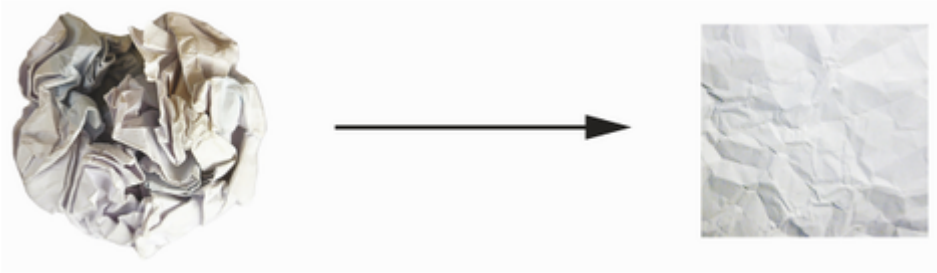
The **ability to interpolate between samples** is the key to understanding generalization in DL.

Interpolation as a source of generalization If you work with data points that can be interpolated, you can start making sense of points you've never seen before by relating them to other points that lie close on the manifold. In other words, you can make sense of the totality of the space using only a sample of the space. You can use interpolation to fill in the blanks.

Interpolation on the latent manifold is different from linear interpolation in the parent space, as illustrated in the figure below. Every point on the latent manifold of digits is a valid digit, but the average of two digits usually isn't.



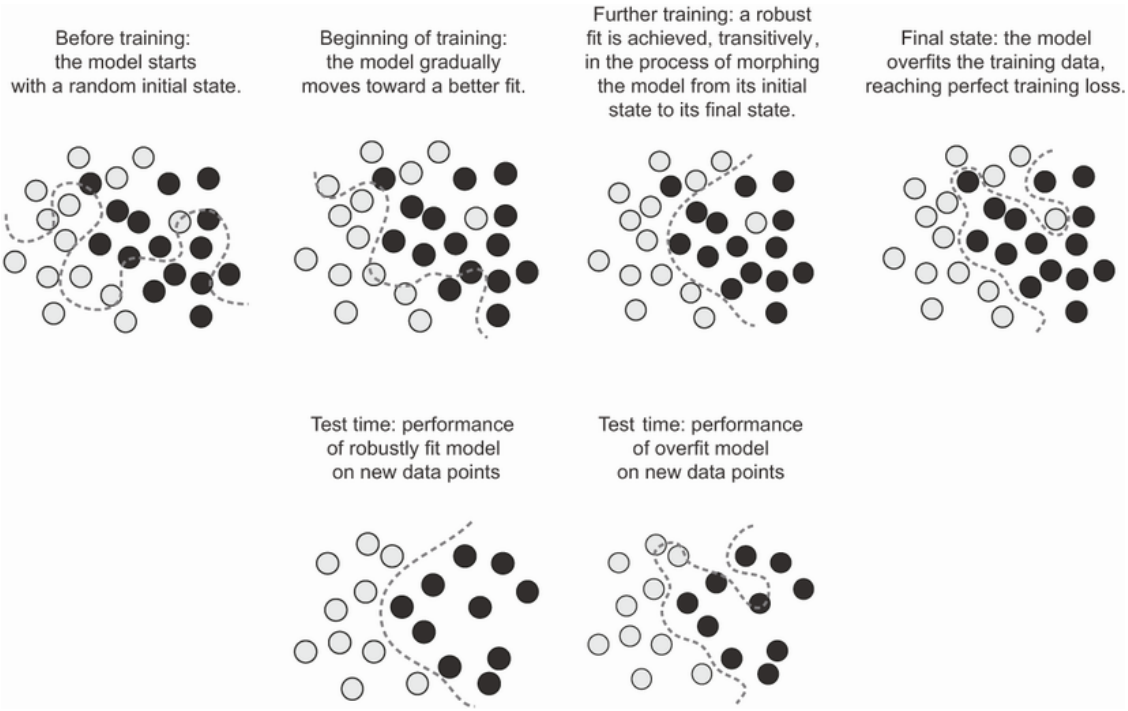
Deep learning and manifolds Imagine two sheets of colored paper: one red and one blue. Put one on top of the other. Now crumple them together into a small ball. That crumpled paper ball is your input data, and each sheet of paper is a class of data in a classification problem. What a neural network is meant to do is figure out a transformation of the paper ball that would uncrumple it, so as to make the two classes cleanly separable again (see the figure below). With DL, this would be implemented as a series of simple transformations of the 3D space, such as those you could apply on the paper ball with your fingers, one movement at a time.



Uncrumpling paper balls is what ML is about: finding representations for complex, highly folded data manifolds in high-dimensional spaces. DL excels at incrementally decomposing a complicated geometric transformation into a long chain of elementary ones, which is pretty much the strategy a human would follow to uncrumple a paper ball. Each layer in a deep network applies a transformation that disentangles the data a little, and a deep stack of layers makes tractable an extremely complicated disentanglement process.

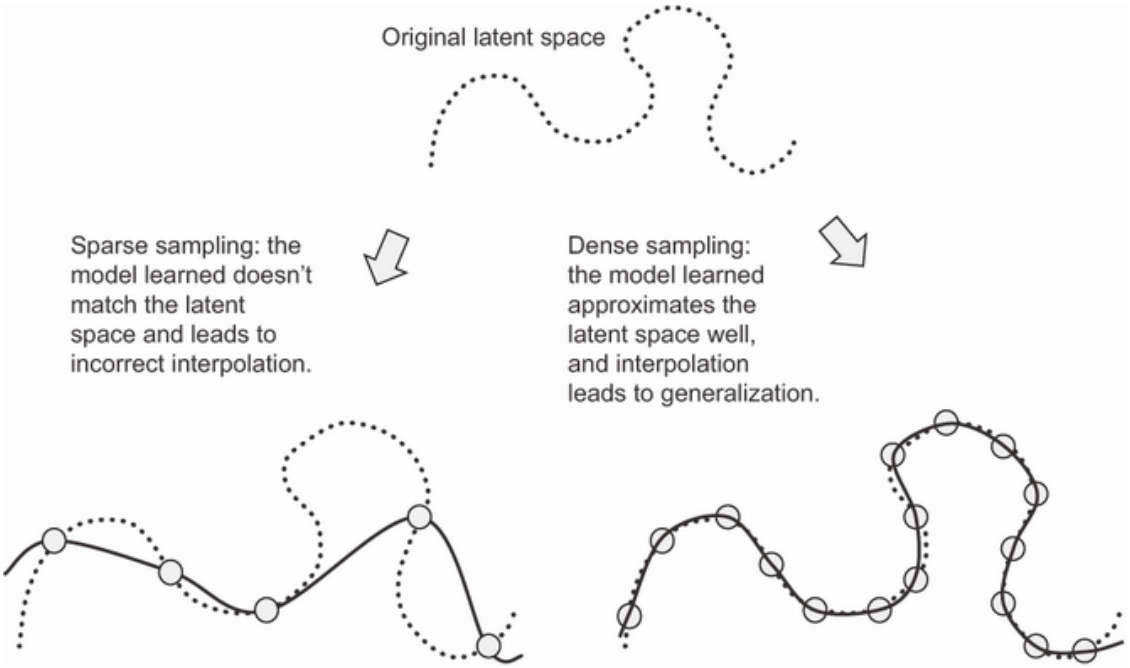
A DL model is basically a very high-dimensional curve — a curve that is smooth and continuous, since it needs to be differentiable. And that curve is fitted to data points via gradient descent, smoothly and incrementally. By its very nature, **DL is about taking a big, complex curve, i.e. a manifold, and incrementally adjusting its parameters until it fits some training data points**.

Fitting our model curve to the data happens gradually and smoothly over time as gradient descent progresses. There is an intermediate point during training at which the model roughly approximates the natural manifold of the data, that is, going from a random model to an overfit model, we achieve a robust fit as an intermediate state. We can see this in the figure below.



Training data is paramount While DL is indeed well suited to manifold learning, the power to generalize is more a consequence of the natural structure of the data than a consequence of any property of our model. We'll be able to generalize only if our data form a manifold where points can be interpolated. The more informative and the less noisy our features are, the better we'll be able to generalize, since our input space will be better structured.

Because DL is curve fitting, for a model to perform well it needs to be trained on a dense sampling of its input space. A "dense sampling" in this context means that the training data should densely cover the entirety of the input data manifold (see figure below). This is especially true near decision boundaries. With a sufficiently dense sampling, it becomes possible to make sense of new inputs by interpolating between past training inputs without having to use common sense, abstract reasoning, or external knowledge about the world — all things that ML models have no access to.



As such, you should always keep in mind that the **best way to improve a DL model is to train it on more data or better data**. Of course, adding overly noisy or inaccurate data will harm generalization. A denser coverage of the input data manifold will yield a model that generalizes better. You should never expect a DL model to perform anything more than crude interpolation between its training samples, and thus you should do everything you can to make interpolation as easy as possible. The only thing you will find in a DL model is what you put into it: the priors encoded in its architecture and the data it was trained on.

When getting more data isn't possible, the next best solution is to **modulate the quantity of information that your model is allowed to store, or to add constraints on the smoothness of the model curve**. If a network can only afford to memorize a small number of patterns, or very regular patterns, the optimization process will force it to focus on the most prominent patterns, which have a better chance of generalizing well. The process of fighting overfitting this way is called **regularization**.

1.2 Evaluating ML models

We can only control what we can observe. Since our goal is to develop models that can successfully generalize to new data, it's essential to be able to reliably measure the generalization power of your model. Hereafter, we'll introduce different ways we can evaluate ML models.

1.2.1 Training, validation and test sets

Evaluating a model always boils down to splitting the available data into three sets: **training**, **validation** and **test**. We train on the training data and evaluate our model on the validation data. Once our model is ready, we test it one final time on the test data, which is meant to be as similar as possible to production data. Then we can deploy the model in production.

Why not have only two sets: a training set and a test set? We'd train on the training data and evaluate on the test data. Much simpler! But…

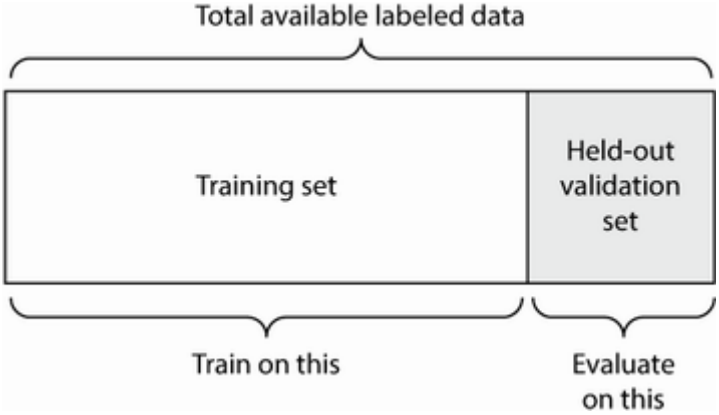
The reason is that developing a model always involves tuning its configuration: for example, choosing the number of layers or the size of the layers (called the hyperparameters of the model, to distinguish them from the parameters, which are the network's weights). We do this tuning by using as a feedback signal the performance of the model on the validation data. In essence, this tuning is a form of learning: a search for a good configuration in some parameter space. As a result, tuning the configuration of the model based on its performance on the validation set can quickly result in overfitting to the validation set, even though your model is never directly trained on it.

Central to this phenomenon is the notion of **information leaks**. Every time we tune a hyperparameter of our model based on the model's performance on the validation set, some information about the validation data leaks into the model. If we do this only once, for one parameter, then very few bits of information will leak, and our validation set will remain reliable for evaluating the model. But if we repeat this many times — running one experiment, evaluating on the validation set, and modifying your model as a result — then we'll leak an increasingly significant amount of information about the validation set into the model.

At the end of the day, we'll end up with a model that performs artificially well on the validation data, because that's what we optimized it for. We care about performance on completely new data, not on the validation data, so we need to use a completely different, never-before-seen dataset to evaluate the model: the test dataset. Our model shouldn't have had access to any information about the test set, even indirectly. If anything about the model has been tuned based on the test set performance, then our measure of generalization will be flawed.

Splitting your data into training, validation, and test sets may seem straightforward, but there are a few advanced ways to do it that can come in handy when **little data is available**. Let's review three classic evaluation recipes: - simple holdout validation, - K-fold validation, and - iterated K-fold validation with shuffling.

Simple hold-out validation Set apart some fraction of the data as the test set. Train on the remaining data, and evaluate on the test set. In order to prevent information leaks, we shouldn't tune your model based on the test set, and therefore we should also reserve a validation set.

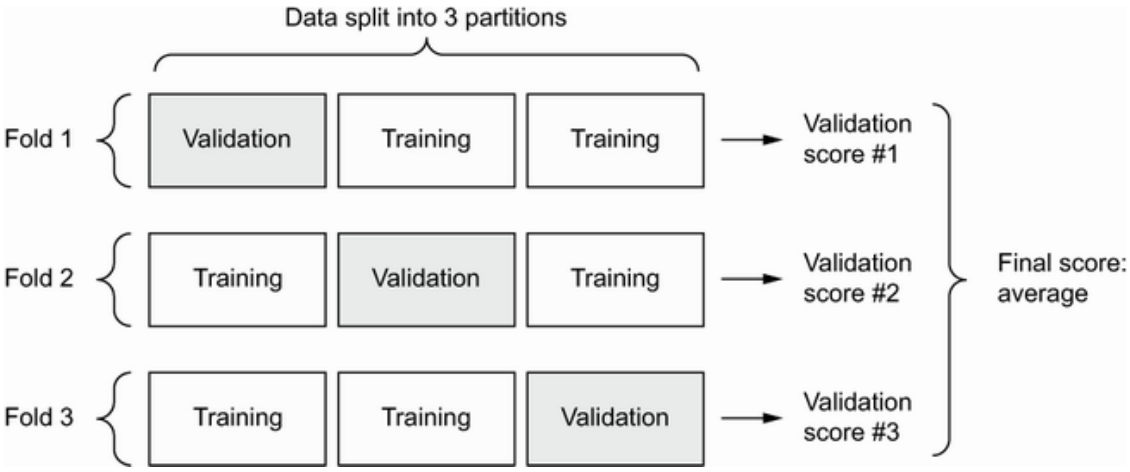


[7]:
```
### Hold-out validation
# num_validation_samples = 10000
# np.random.shuffle(data)
```

```
# validation_data = data[:num_validation_samples]
# training_data = data[num_validation_samples:]
# model = get_model()
# model.fit(training_data, ...)
# validation_score = model.evaluate(validation_data, ...)

# ...

# model = get_model()
# model.fit(np.concatenate([training_data, validation_data]), ...)
# test_score = model.evaluate(test_data, ...)
```

This is the simplest evaluation protocol, and it suffers from one flaw: if little data is available, our validation and test sets may contain too few samples to be statistically representative of the data at hand. This is easy to recognize: if different random shuffling rounds of the data before splitting end up yielding very different measures of model performance, then we're having this issue. **K-fold validation** and **iterated K-fold validation** address this issue.

K-fold validation With K-fold validation, we split our data into K partitions of equal size. For each partition `i`, train a model on the remaining K - 1 partitions, and evaluate it on partition `i`. Our final score is the average of the `K` scores obtained. This method is helpful when the performance of your model shows significant variance based on your train-test split.



[8]:
```
### K-fold validation
# k = 3
# num_validation_samples = len(data) // k
# np.random.shuffle(data)
# validation_scores = []
# for fold in range(k):
#     validation_data = data[num_validation_samples * fold:
#                            num_validation_samples * (fold + 1)]
#     training_data = np.concatenate(
#         data[:num_validation_samples * fold],
```

```
#         data[num_validation_samples * (fold + 1):])
#     model = get_model()
#     model.fit(training_data, ...)
#     validation_score = model.evaluate(validation_data, ...)
#     validation_scores.append(validation_score)
# validation_score = np.average(validation_scores)
# model = get_model()
# model.fit(data, ...)
# test_score = model.evaluate(test_data, ...)
```

Iterated K-fold validation with shuffling This one is for situations in which we have relatively little data available and we need to evaluate our model as precisely as possible. It is very helpful in Kaggle competitions. It consists of applying K-fold validation multiple times, shuffling the data every time before splitting it K ways. The final score is the average of the scores obtained at each run of K-fold validation.

Note that we end up training and evaluating P K* models (where P is the number of iterations you use), which can be very expensive.

1.3 Improving model fit

To achieve the perfect fit, we must first overfit. Since we don't know in advance where the boundary lies, we must cross it to find it. Thus, our initial goal as we start working on a problem is to achieve a model that shows some generalization power and that is able to overfit. Once we have such a model, we'll focus on refining generalization by fighting overfitting.

There are three common problems encountered at this stage: - Training doesn't get started: our training loss doesn't decrease with epochs. - Training gets started just fine, but our model doesn't meaningfully generalize. - Training and validation loss both go down over time, but we don't seem to be able to overfit, which indicates we're still underfitting.

Let's see how you can address these issues to achieve the first big milestone of an ML project: *getting a model that has some generalization power and that is able to overfit.*

1.3.1 Tuning key gradient descent parameters

Sometimes training doesn't get started, or it stalls too early. The loss is stuck. This is something we can overcome: remember that we can fit a model to random data. Even if nothing about our problem makes sense, we should still be able to train something — if only by memorizing the training data.

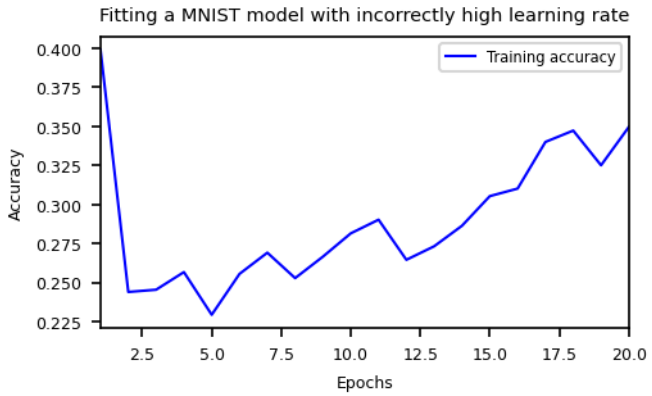
When this happens, it's always a problem with the configuration of the gradient descent process: our choice of optimizer, the distribution of initial values in the weights of your model, learning rate, or batch size. All these parameters are interdependent, and it is usually sufficient to tune the learning rate and the batch size while keeping the rest of the parameters constant.

Let's look at a concrete example: let's train the MNIST model with an inappropriately large learning rate of value 1.

Training a MNIST model with an incorrectly high learning rate

[9]:
```
(train_images, train_labels), _ = mnist.load_data()
train_images = train_images.reshape((60000, 28 * 28))
train_images = train_images.astype("float32") / 255

model = keras.Sequential([
    layers.Dense(512, activation="relu"),
    layers.Dense(10, activation="softmax")
])
model.compile(optimizer=keras.optimizers.RMSprop(1.),
              loss="sparse_categorical_crossentropy",
              metrics=["accuracy"])
history_high_LR = model.fit(train_images, train_labels,
                            epochs=20,
                            batch_size=128,
                            validation_split=0.2,
                            verbose=0)
```

[10]:
```
plot_helper([range(1, len(history_high_LR.history["accuracy"]) + 1)],
            [history_high_LR.history["accuracy"]],
            ["b-"],
            ["Training accuracy"],
            "Fitting a MNIST model with incorrectly high learning rate",
            "Epochs",
            "Accuracy")
```

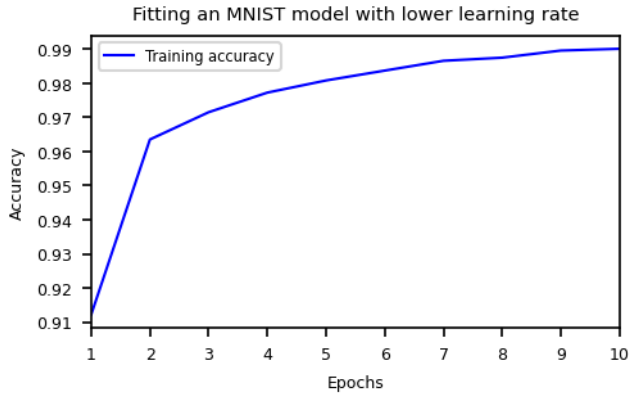


The model quickly reaches a training and validation accuracy in the 20%–40% range, but cannot get past that. Let's try to lower the learning rate to a more reasonable value of 1e-2.

[11]:
```
model = keras.Sequential([
    layers.Dense(512, activation="relu"),
    layers.Dense(10, activation="softmax")
])
model.compile(optimizer=keras.optimizers.RMSprop(1e-2),
```

```
              loss="sparse_categorical_crossentropy",
              metrics=["accuracy"])
history_low_LR = model.fit(train_images, train_labels,
                           epochs=10,
                           batch_size=128,
                           validation_split=0.2,
                           verbose=0)
```

[12]:
```
plot_helper([range(1, len(history_low_LR.history["accuracy"]) + 1)],
            [history_low_LR.history["accuracy"]],
            ["b-"],
            ["Training accuracy"],
            "Fitting an MNIST model with lower learning rate",
            "Epochs",
            "Accuracy")
```



So, in a situation like this one, we can try: - Lowering or increasing the learning rate. A learning rate that is too high may lead to updates that vastly overshoot a proper fit, like in the preceding example, and a learning rate that is too low may make training so slow that it appears to stall. - Increasing the batch size. A batch with more samples will lead to gradients that are more informative and less noisy (lower variance).

1.3.2 Leveraging better architecture priors

We have a model that fits the training data, but for some reason your validation metrics aren't improving at all. They remain no better than what a random classifier would achieve: our model trains but doesn't generalize. What's going on?

This is perhaps the **worst ML situation** we can find ourselves in. It indicates that something is fundamentally wrong with our approach, and it may not be easy to tell what. Here are some tips.

First, it may be that the input data we're using simply doesn't contain sufficient information to predict our targets: the problem as formulated is not solvable. This is what happened earlier when we tried to fit an MNIST model where the labels were shuffled: the model would train just fine,

but validation accuracy would stay stuck at 10%, because it was plainly impossible to generalize with such a dataset.

It may also be that the kind of model we're using is not suited for the problem at hand. Using a model that makes the right assumptions about the problem is essential to achieve generalization: we should leverage the right architecture priors. For example, for image classification, a 2D convolutional NN should be used.

1.3.3 Increasing model capacity

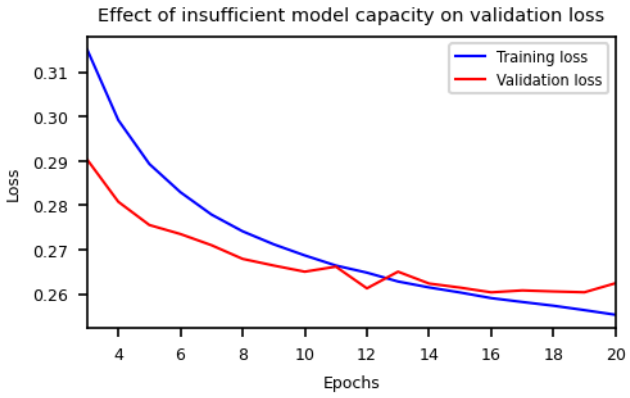
If we manage to get to a model that fits, and validation metrics are going down, and that seems to achieve at least some level of generalization power - we're almost there! Next, we need to get our model to **start overfitting**.

A simple logistic regression on MNIST

Consider the following small model — a simple logistic regression — trained on MNIST.

[13]:
```
model = keras.Sequential([layers.Dense(10, activation="softmax")])
model.compile(optimizer="rmsprop",
              loss="sparse_categorical_crossentropy",
              metrics=["accuracy"])
history_small_model = model.fit(
    train_images, train_labels,
    epochs=20,
    batch_size=128,
    validation_split=0.2,
    verbose=0)
```

[14]:
```
loss = history_small_model.history["loss"]
val_loss = history_small_model.history["val_loss"]
epochs = range(3, len(loss) + 1)
plot_helper([epochs, epochs],
            [loss[2:], val_loss[2:]],
            ["b-", "r-"],
            ["Training loss", "Validation loss"],
            "Effect of insufficient model capacity on validation loss",
            "Epochs",
            "Loss")
```



Validation loss stagnates, or improves very slowly, instead of peaking and reversing course (kind of U-shape). The validation loss goes to 0.26 and just stays there. You can fit (training loss goes down over epochs), but we can't overfit, even after many iterations over the training data.

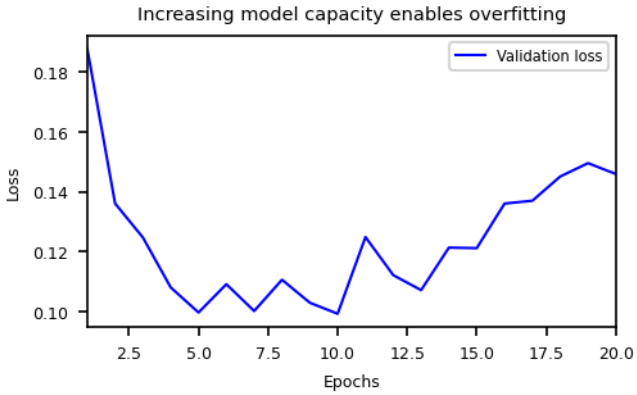
It should always be possible to overfit! Much like the problem where the training loss doesn't go down, this is an issue that can always be solved. If we're not able to overfit, it's likely a problem with the **representational power** of the model: we're going to need a bigger model, one with more capacity - one able to store more information and to fit a wider variety of functions. We can increase representational power by adding more layers, using bigger layers (layers with more parameters), or using kinds of layers that are more appropriate for the problem at hand (better architecture priors).

Let's try training a bigger model, one with two intermediate layers with 96 units each:

[15]:
```
model = keras.Sequential([
    layers.Dense(96, activation="relu"),
    layers.Dense(96, activation="relu"),
    layers.Dense(10, activation="softmax"),
])
model.compile(optimizer="rmsprop",
              loss="sparse_categorical_crossentropy",
              metrics=["accuracy"])
history_large_model = model.fit(
    train_images, train_labels,
    epochs=20,
    batch_size=128,
    validation_split=0.2,
    verbose=0)
```

[16]:
```
loss = history_large_model.history["loss"]
val_loss = history_large_model.history["val_loss"]
epochs = range(1, 21)
plot_helper([epochs],
            [val_loss],
```

```
            ["b"],
            ["Validation loss"],
            "Increasing model capacity enables overfitting",
            "Epochs",
            "Loss")
```



1.4 Improving generalization

Once our model has demonstrated generalization power and capability to overfit, it's time to switch our focus to maximizing generalization.

1.4.1 Dataset curation

We've already learned that generalization in DL originates from the latent structure of your data. If our data makes it possible to smoothly interpolate between samples, we'll be able to train a DL model that generalizes. If our problem is overly noisy or fundamentally discrete, DL will not help us. DL is just curve fitting, not magic.

It is essential that we're working with an appropriate dataset. **Spending more effort and money on data collection almost always yields a much greater return on investment than spending the same on developing a better model**.

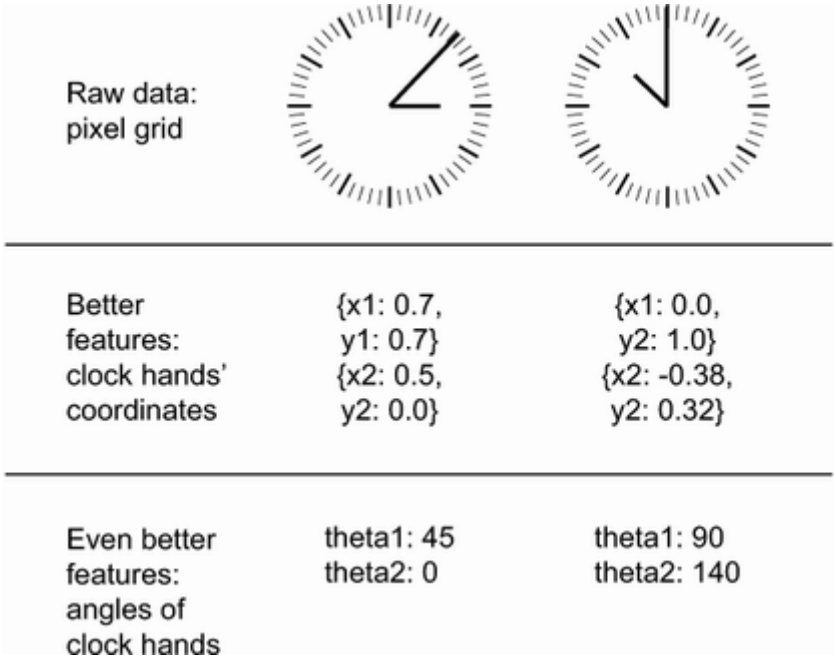
- Make sure you have enough data. Remember that we need a dense sampling of the input-cross-output space. More data will yield a better model. Sometimes, problems that seem impossible at first become solvable with a larger dataset.
- Minimize labeling errors — visualize your inputs to check for anomalies, and proofread your labels.
- Clean your data and deal with missing values.
- If we have many features and we aren't sure which ones are actually useful, do feature selection.

A particularly important way to improve the generalization potential of your data is **feature engineering**. For most ML problems, feature engineering is a key ingredient for success.

1.4.2 Feature engineering

Feature engineering is the process of using our own knowledge about the data and the ML algorithm at hand (in this case, a neural network) to make the algorithm work better by applying hardcoded (non-learned) transformations to the data before it goes into the model. In many cases, it isn't reasonable to expect an ML 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.

Let's look at an intuitive example. Suppose we're trying to develop a model that can take as input an image of a clock and can output the time of day.



If we choose to use the raw pixels of the image as input data, we'll have a difficult ML problem. We'll need a convolutional neural network to solve it, and we'll have to expend quite a bit of computational resources to train the network.

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

We can go even further: do a coordinate change, and express the (x, y) coordinates of the tip of each hand as polar coordinates with regard to the center of the image. Our input will become the angle theta of each clock hand. At this point, our features are making the problem so easy that no ML is required; a simple rounding operation and dictionary lookup are enough to recover the approximate time of day.

That's the essence of feature engineering: **making a problem easier by expressing it in a simpler way**. Make the latent manifold smoother, simpler, better organized. Doing so usually requires understanding the problem in depth.

Before DL, feature engineering used to be the most important part of the ML workflow, because classical shallow algorithms didn't have hypothesis spaces rich enough to learn useful features by themselves. The way we presented the data to the algorithm was absolutely critical to its success. For instance, before CNNs became successful on the MNIST digit-classification problem, solutions were typically based on hardcoded features such as the number of loops in a digit image, the height of each digit in an image, a histogram of pixel values, and so on.

Fortunately, modern DL removes the need for most feature engineering, because neural networks are capable of automatically extracting useful features from raw data. Does this mean we don't have to worry about feature engineering as long as we're using deep neural networks? No, for two reasons: - Good features still allow us to solve problems more elegantly while using fewer resources. For instance, it would be ridiculous to solve the problem of reading a clock face using a CNN. - Good features let us solve a problem with far less data. The ability of DL models to learn features on their own relies on having lots of training data available; if we have only a few samples, the information value in their features becomes critical.

1.4.3 Using early stopping

In DL, it's common to use models that are greatly **overparameterized**: they have many more degrees of freedom (i.e. capacity) than the minimum necessary to fit to the latent manifold of the data. This overparameterization is not an issue, because we never fully fit a DL model. Such a fit wouldn't generalize at all. We will **always interrupt training long before the minimum possible training loss is reached**.

Finding the exact point during training where we've reached the most generalizable fit — the exact boundary between an underfit curve and an overfit curve — is one of the most effective things we can do to improve generalization.

In the previous examples, we would start by training our models for longer than needed to figure out the number of epochs that yielded the best validation metrics, and then we would retrain a new model for exactly that number of epochs. This is pretty standard, but it requires you to do redundant work, which can sometimes be expensive. Naturally, you could just save your model at the end of each epoch, and once you've found the best epoch, reuse the closest saved model you have.

In Keras, it's typical to do this with an `EarlyStopping` callback, which will interrupt training as soon as validation metrics have stopped improving, while remembering the best known model state. We'll learn how to use callbacks in later lectures.

1.4.4 Regularizing a model

Regularization techniques are a set of best practices that actively impede the model's ability to fit perfectly to the training data, with the goal of making the model perform better during validation. This is called **regularizing** the model, because it tends to make the model simpler, more "regular," its curve smoother, more "generic". This way, we develop a model less specific to the training set and better able to generalize by more closely approximating the latent manifold of the data.

Let's review some of the most common regularization techniques and apply them in practice to improve the IMDB movie-classification model.

Reducing the network's size A model that is too small will not overfit. The simplest way to mitigate overfitting is to reduce the size of the model (the number of learnable parameters in the model, determined by the number of layers and the number of units per layer). If the model has limited memorization resources, it won't be able to simply memorize its training data. Therefore, in order to minimize its loss, it will have to resort to learning compressed representations that have predictive power regarding the targets — precisely the type of representations we're interested in. At the same time, we should use models that have enough parameters that they don't underfit. There is a compromise to be found between too much capacity and not enough capacity.

Unfortunately, there is no magical formula to determine the right number of layers or the right size for each layer. We must evaluate an array of different architectures (using the validation set) in order to find the correct model size for our data. The general workflow for finding an appropriate model size is to **start with relatively few layers and parameters, and increase the size of the layers or add new layers until we see very little improvement in validation loss**.

Let's try this on the IMDB movie-review classification model. The following listing shows our original model.

Original model

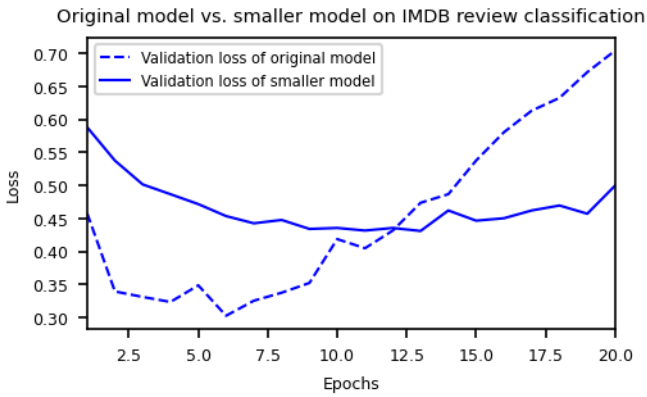
[17]:
```
from tensorflow.keras.datasets import imdb
(train_data, train_labels), _ = imdb.load_data(num_words=10000)

def vectorize_sequences(sequences, dimension=10000):
    results = np.zeros((len(sequences), dimension))
    for i, sequence in enumerate(sequences):
        results[i, sequence] = 1.
    return results
train_data = vectorize_sequences(train_data)

model = keras.Sequential([
    layers.Dense(16, activation="relu"),
    layers.Dense(16, activation="relu"),
    layers.Dense(1, activation="sigmoid")
])
model.compile(optimizer="rmsprop",
              loss="binary_crossentropy",
              metrics=["accuracy"])
history_original = model.fit(train_data, train_labels,
                             epochs=20, batch_size=512, validation_split=0.4,␣
 ↪verbose=0)
```

Now let's try to replace it with a smaller model (model with lower capacity)

[18]:
```
model = keras.Sequential([
    layers.Dense(4, activation="relu"),
    layers.Dense(4, activation="relu"),
    layers.Dense(1, activation="sigmoid")
])
```

```
model.compile(optimizer="rmsprop",
              loss="binary_crossentropy",
              metrics=["accuracy"])
history_smaller_model = model.fit(
    train_data, train_labels,
    epochs=20, batch_size=512, validation_split=0.4, verbose=0)
```

[19]:
```
val_loss_original = history_original.history["val_loss"]
val_loss_smaller = history_smaller_model.history["val_loss"]
epochs = range(1, len(val_loss_original) + 1)
plot_helper([epochs, epochs],
            [val_loss_original, val_loss_smaller],
            ["b--", "b"],
            ["Validation loss of original model", "Validation loss of smaller␣
 ↪model"],
            "Original model vs. smaller model on IMDB review classification",
            "Epochs",
            "Loss")
```



As we can see, the smaller model starts overfitting later than the reference model (after six epochs rather than four), and its performance degrades more slowly once it starts overfitting.

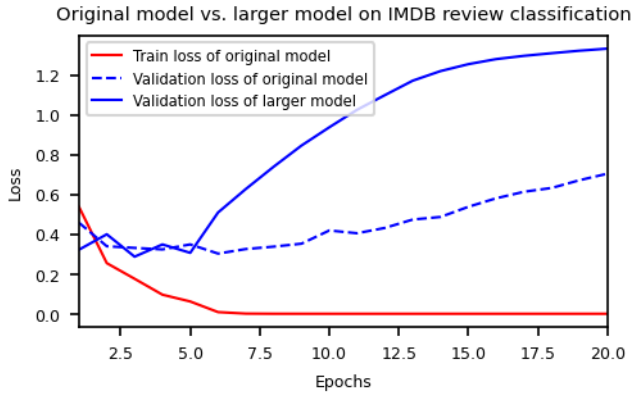
Now, let's add a model that has much more capacity — far more than the problem requires. It is standard to work with models that are significantly overparameterized for what they're trying to learn, i.e., models with too much memorization capacity. We'll know our **model is too large if it starts overfitting right away and if its validation loss curve looks choppy with high-variance** (although choppy validation metrics could also be a symptom of using an unreliable validation process, such as a validation split that's too small).

Version of the model with much higher capacity

[20]:
```
model = keras.Sequential([
    layers.Dense(512, activation="relu"),
```

```
    layers.Dense(512, activation="relu"),
    layers.Dense(1, activation="sigmoid")
])
model.compile(optimizer="rmsprop",
              loss="binary_crossentropy",
              metrics=["accuracy"])
history_larger_model = model.fit(
    train_data, train_labels,
    epochs=20, batch_size=512, validation_split=0.4, verbose=0)
```

[21]:
```
train_loss_larger = history_larger_model.history["loss"]
val_loss_larger = history_larger_model.history["val_loss"]
epochs = range(1, len(val_loss_original) + 1)
plot_helper([epochs, epochs, epochs],
            [train_loss_larger, val_loss_original, val_loss_larger],
            ["r", "b--", "b"],
            ["Train loss of original model", "Validation loss of original␣
 ↪model", "Validation loss of larger model"],
            "Original model vs. larger model on IMDB review classification",
            "Epochs",
            "Loss")
```



The bigger model starts overfitting almost immediately, after just one-two epochs, and it overfits much more severely. Its validation loss is also noisier. It gets training loss near zero very quickly. The more capacity the model has, the more quickly it can model the training data (resulting in a low training loss), but the more susceptible it is to overfitting (resulting in a large difference between the training and validation loss).

Adding weight regularization You may be familiar with the principle of **Occam's razor**: given two explanations for something, the explanation most likely to be correct is the simplest one — the one that makes fewer assumptions. This idea also applies to the models learned by neural networks: given some training data and a network architecture, multiple sets of weight values

(multiple models) could explain the data. Simpler models are less likely to overfit than complex ones.

A simple model in this context is a model where the distribution of parameter values has less entropy (or a model with fewer parameters). Thus, a common way to mitigate overfitting is to put constraints on the complexity of a model 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 model a cost associated with having large weights. Weight regularization 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.

L1 regularization on least squares:

$$\mathbf{w}^* = \arg\min_{\mathbf{w}} \sum_j \left(t(\mathbf{x}_j) - \sum_i w_i h_i(\mathbf{x}_j) \right)^2 + \lambda \sum_{i=1}^{k} |w_i|$$

L2 regularization on least squares:

$$\mathbf{w}^* = \arg\min_{\mathbf{w}} \sum_j \left(t(\mathbf{x}_j) - \sum_i w_i h_i(\mathbf{x}_j) \right)^2 + \lambda \sum_{i=1}^{k} w_i^2$$

In Keras, weight regularization is added by passing weight regularizer instances to layers as keyword arguments. Let's add L2 weight regularization to our initial movie-review classification model.

Adding L2 weight regularization to the model

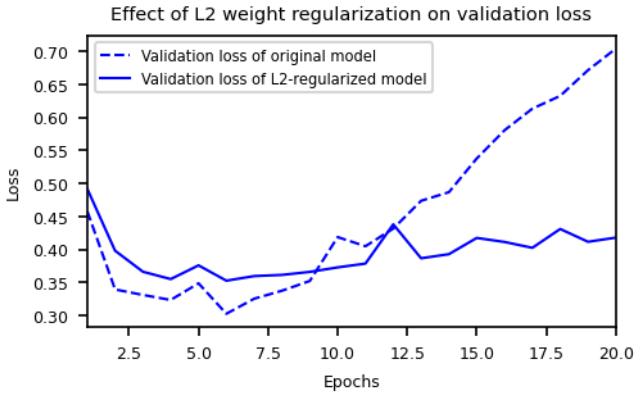
[22]:
```
from tensorflow.keras import regularizers
model = keras.Sequential([
    layers.Dense(16,
                 kernel_regularizer=regularizers.l2(0.002),
                 activation="relu"),
    layers.Dense(16,
                 kernel_regularizer=regularizers.l2(0.002),
                 activation="relu"),
    layers.Dense(1, activation="sigmoid")
])
model.compile(optimizer="rmsprop",
              loss="binary_crossentropy",
              metrics=["accuracy"])
history_l2_reg = model.fit(
```

```
    train_data, train_labels,
    epochs=20, batch_size=512, validation_split=0.4, verbose=0)
```

In the preceding listing, `l2(0.002)` means every coefficient in the weight matrix of the layer will add `0.002 * weight_coefficient_value ** 2` to the total loss of the model. Because this penalty is only added at training time, the loss for this model will be much higher at training than at test time.

The figure below shows the impact of the L2 regularization penalty. The model with L2 regularization has become much more resistant to overfitting than the reference model, even though both models have the same number of parameters.

[23]:
```
val_loss_original = history_original.history["val_loss"]
val_loss_l2_reg = history_l2_reg.history["val_loss"]
epochs = range(1, len(val_loss_original) + 1)
plot_helper([epochs, epochs],
            [val_loss_original, val_loss_l2_reg],
            ["b--", "b"],
            ["Validation loss of original model", "Validation loss of␣
 ↪L2-regularized model"],
            "Effect of L2 weight regularization on validation loss",
            "Epochs",
            "Loss")
```



Different weight regularizers available in Keras

As an alternative to L2 regularization, we can use L1 regularization or combined L1-L2 regularization:

```
regularizers.l1(0.001)
```

```
regularizers.l1_l2(l1=0.001, l2=0.001)
```

Adding dropout Dropout is one of the most effective and most commonly used regularization techniques for neural networks. Dropout, applied to a layer, consists of randomly **dropping out**

(setting to zero) a number of output features of the layer during training. Let's say a given layer would normally return a vector [0.2, 0.5, 1.3, 0.8, 1.1] for a given input sample during training. After applying dropout, this vector will have a few zero entries distributed at random: for example, [**0**, 0.5, **0**, 0.8, 1.1]. The **dropout rate** is the fraction of the features that are zeroed out, usually set between 0.2 and 0.5.

Dropout is not used after training when making a prediction (inference), i.e., at test time no units are dropped out. The weights of the network will be larger than normal because of dropout. Therefore, before finalizing the network, the weights are first scaled by the chosen dropout rate.

Consider a NumPy matrix containing the output of a layer, layer_output, of shape `(batch_size, features)`. At training time, we zero out at random a fraction of the values in the matrix:

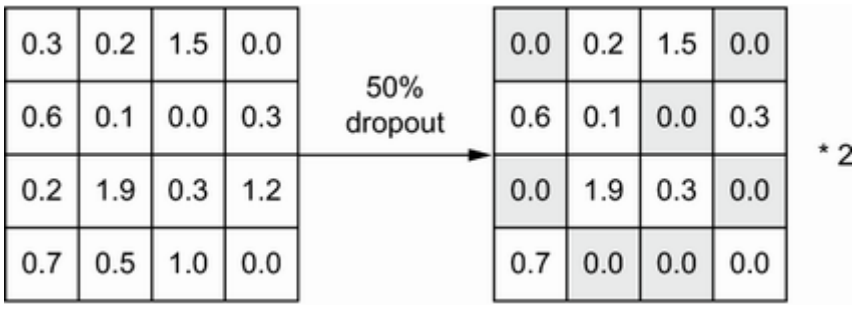
```
layer_output *= np.random.randint(0, high=2, size=layer_output.shape)
```

At test time, we scale down the output by the dropout rate. Here, we scale by 0.5 (because we previously dropped half the units):

```
layer_output *= 0.5
```

The rescaling of the weights can be performed at training time instead, after each weight update at the end of the mini-batch. This is sometimes called **inverse dropout** and does not require any modification of weights during testing. This is often the way it's implemented in practice (see figure below). Keras implements dropout in this way. This process can be implemented by doing both operations at training time and leaving the output unchanged at test time, which is:

```
layer_output *= np.random.randint(0, high=2, size=layer_output.shape)
layer_output /= 0.5
```



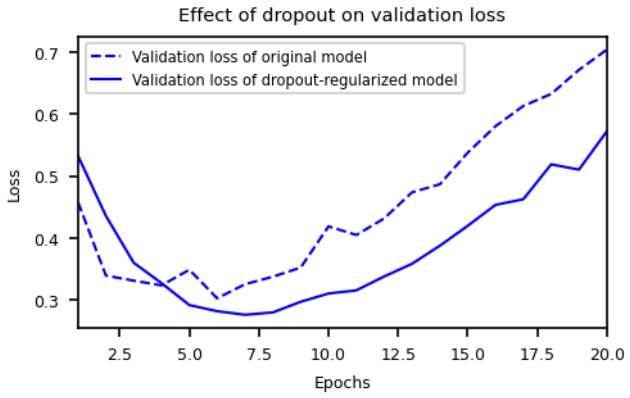
This technique may seem strange and arbitrary. Why would this help reduce overfitting? Geoff Hinton (inventor of dropout) says he was inspired by, among other things, a fraud-prevention mechanism used by banks. In his own words, "I went to my bank. The tellers kept changing and I asked one of them why. He said he didn't know but they got moved around a lot. I figured it must be because it would require cooperation between employees to successfully defraud the bank. This made me realize that randomly removing a different subset of neurons on each example would prevent conspiracies and thus reduce overfitting."

In Keras, we can introduce dropout in a model via the `Dropout` layer, which is applied to the output of the layer right before it. Let's add two `Dropout` layers in the IMDB model to see how well they do at reducing overfitting.

Adding dropout to the IMDB model

[24]:
```
model = keras.Sequential([
    layers.Dense(16, activation="relu"),
    layers.Dropout(0.5),
    layers.Dense(16, activation="relu"),
    layers.Dropout(0.5),
    layers.Dense(1, activation="sigmoid")
])
model.compile(optimizer="rmsprop",
              loss="binary_crossentropy",
              metrics=["accuracy"])
history_dropout = model.fit(
    train_data, train_labels,
    epochs=20, batch_size=512, validation_split=0.4, verbose=0)
```

[25]:
```
val_loss_original = history_original.history["val_loss"]
val_loss_dropout = history_dropout.history["val_loss"]
epochs = range(1, len(val_loss_original) + 1)
plot_helper([epochs, epochs],
            [val_loss_original, val_loss_dropout],
            ["b--", "b"],
            ["Validation loss of original model", "Validation loss of␣
 ↪dropout-regularized model"],
            "Effect of dropout on validation loss",
            "Epochs",
            "Loss")
```



This is a clear improvement over the reference model — it also seems to be working much better than L2 regularization, since the lowest validation loss reached has improved.

To recap, these are the most common ways to maximize generalization and prevent overfitting in neural networks:

- Get more training data, or better training data.
- Develop better features.

- Reduce the capacity of the model.
- Add weight regularization (for smaller models).
- Add dropout.