Key Issues in Machine Learning

Outline

- Understanding the tension between generalization and optimization, the fundamental issue in deep learning
- Evaluation methods for machine learning models
- Best practices to improve model fitting
- Best practices to achieve better generalization

Generalization:
The goal of machine learning

Generalization: The goal of machine learning

- *Optimization* refers to the process of adjusting a model to get the best performance possible on the training data
- Generalization refers to how well the trained model performs on data it has never seen before.

Generalization: The goal of machine learning

- The goal of the game is to get good generalization,
- 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.

Underfitting and Overfitting

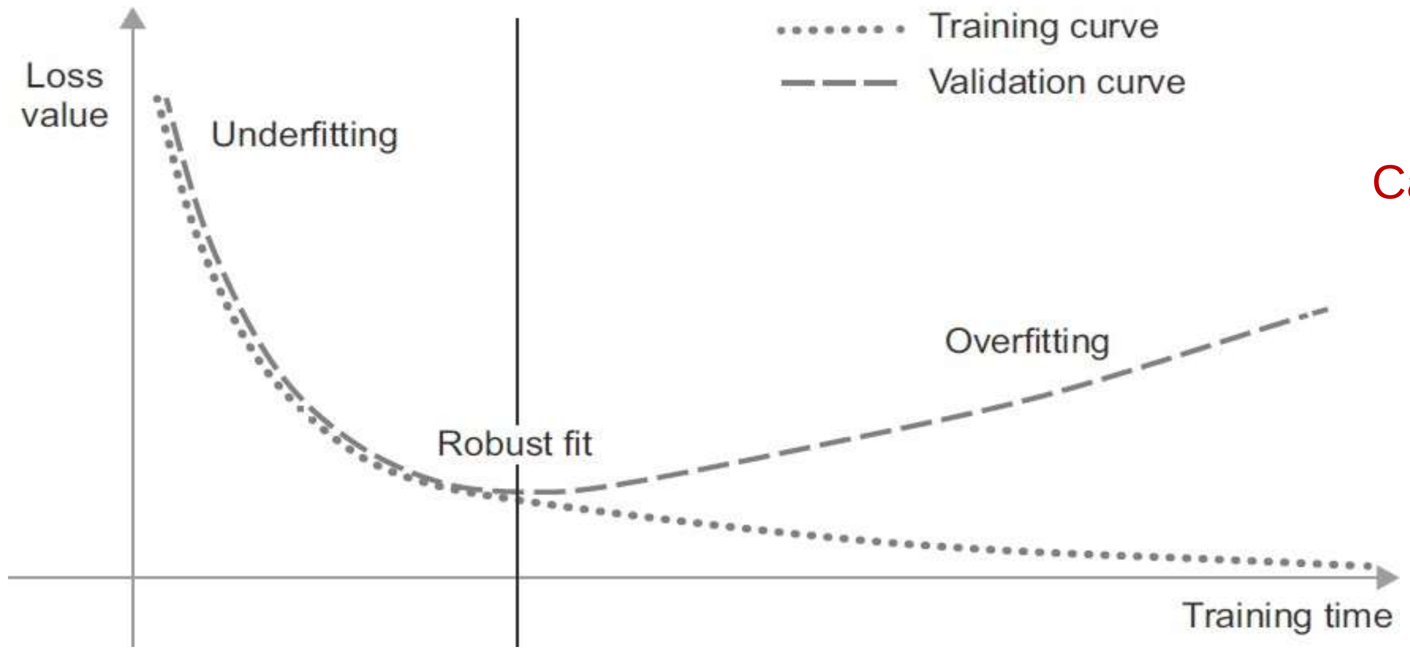
- At the beginning of training, optimization and generalization are correlated: the lower the loss on training data, the lower the loss on test data.
- While this is happening, your model is said to be ***underfit:*** there is still progress to be made.
- At this point, the network hasn't yet modeled all relevant patterns in the training data.

Underfitting and Overfitting

- 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.**
- 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.

Underfitting and Overfitting…

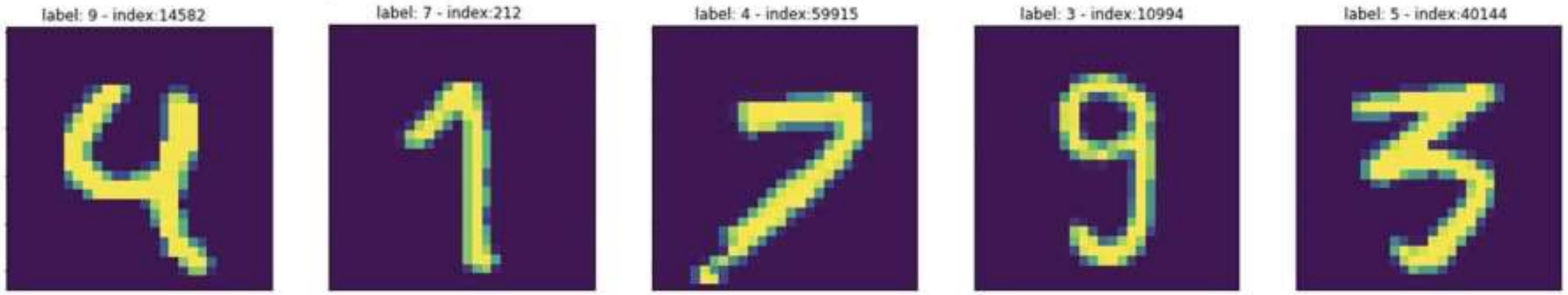
- Performance on the held-out validation data started improving as training went on and then inevitably peaked after a while.
- This pattern is universal.
- This pattern is observed with any model type and any dataset.



Canonical overfitting behavior

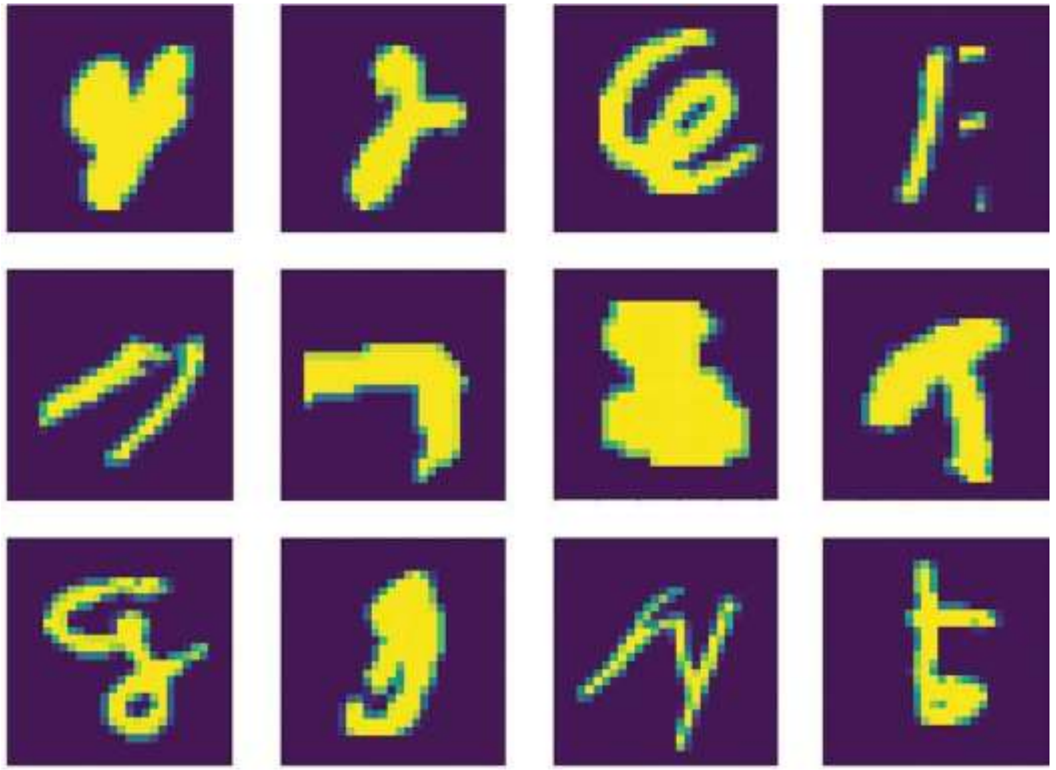
Underfitting and Overfitting…

- Overfitting occurs when the **data**
    - is noisy,
    - involves uncertainty,
    - includes rare features.

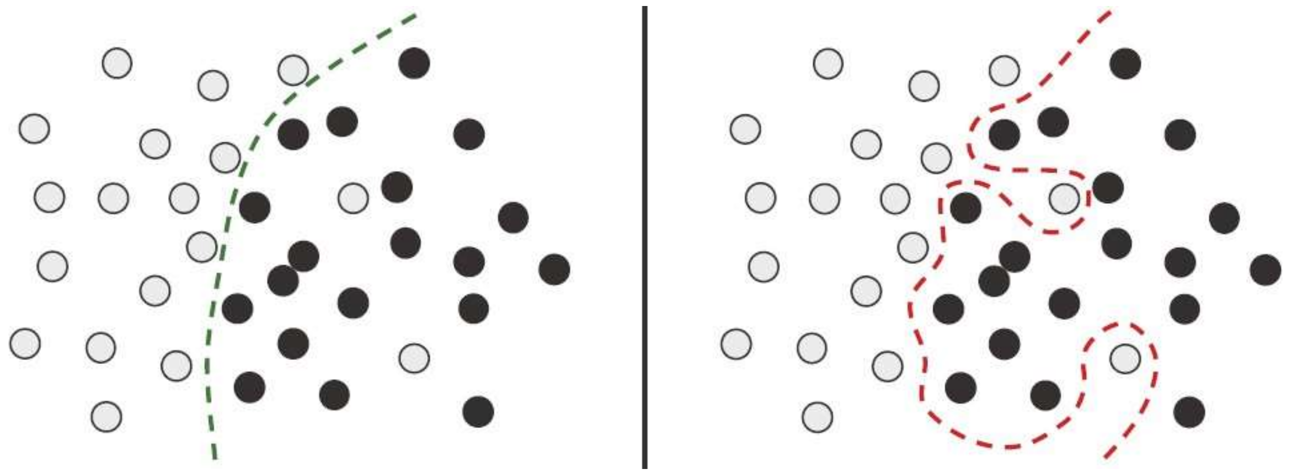


Label: 9 Label: 7 Label: 4 Label: 3 Label: 5

Noisy data



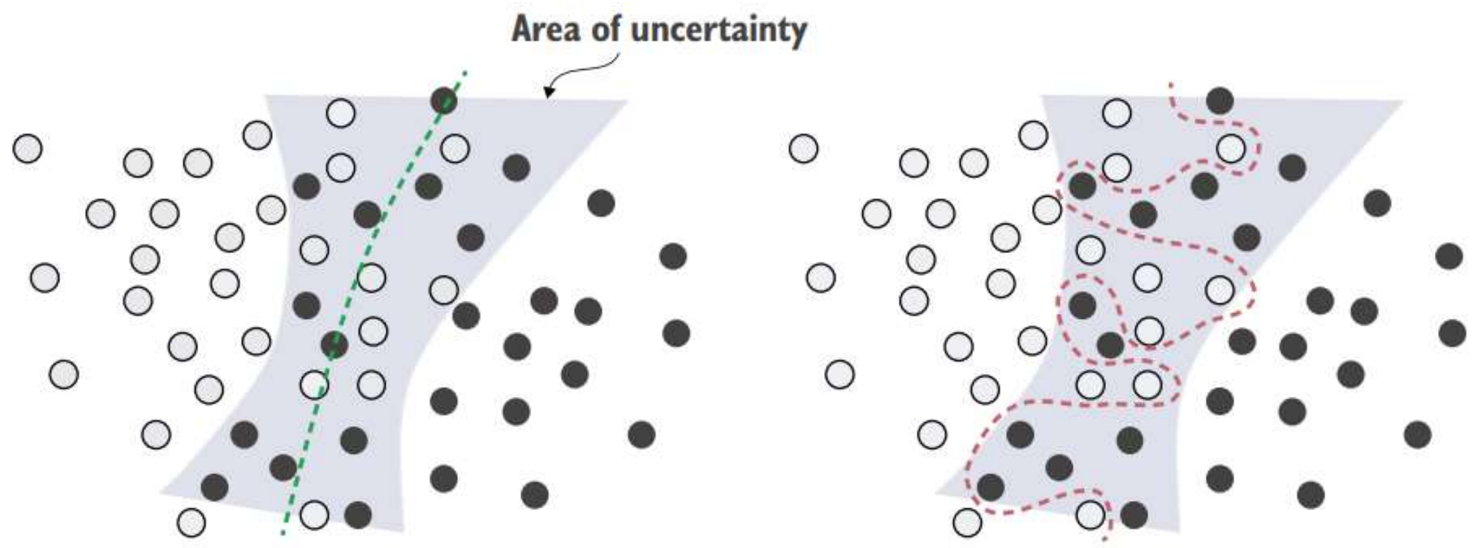
Robust fit vs. overfitting - Outliers



Dealing with outliers: robust vs overfitting

- If a model goes out of its way to incorporate such outliers, its generalization performance will degrade.

Robust fit vs. overfitting- Ambiguous Features



Robust fit vs. overfitting giving an ambiguous area of the feature space

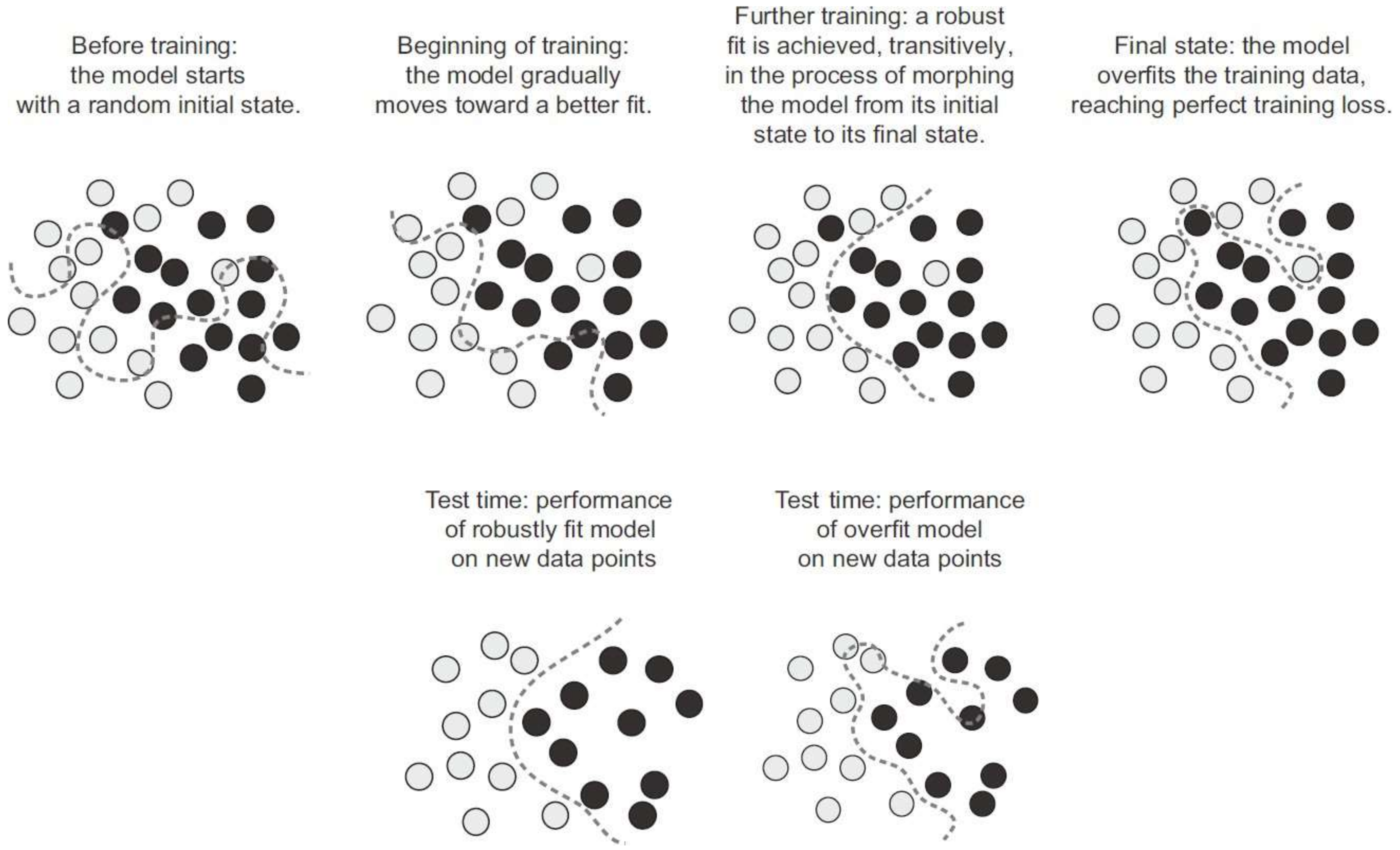
- A model could overfit to such probabilistic data by being too confident about ambiguous regions of the feature space
- A more robust fit would ignore individual data points and look at the bigger picture.

Robust fit vs. overfitting- Rare Features

- Machine learning models trained on datasets that include rare feature values are highly susceptible to overfitting
- 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 —and only keep features that are above some threshold.

The nature of generalization in deep learning

- A remarkable fact about deep learning models is that they can be trained to fit anything, as long as they have enough representational power.
- A deep learning 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, deep learning is about taking a big, complex curve—a manifold—and incrementally adjusting its parameters until it fits some training data points.
- If you let your model train for long enough, it will effectively end up purely memorizing its training data and won't generalize at all!


Before training: the model starts with a random initial state.
Beginning of training: the model gradually moves toward a better fit.
Further training: a robust fit is achieved, transitively, in the process of morphing the model from its initial state to its final state.
Final state: the model overfits the training data, reaching perfect training loss.
Test time: performance of robustly fit model on new data points
Test time: performance of overfit model on new data points

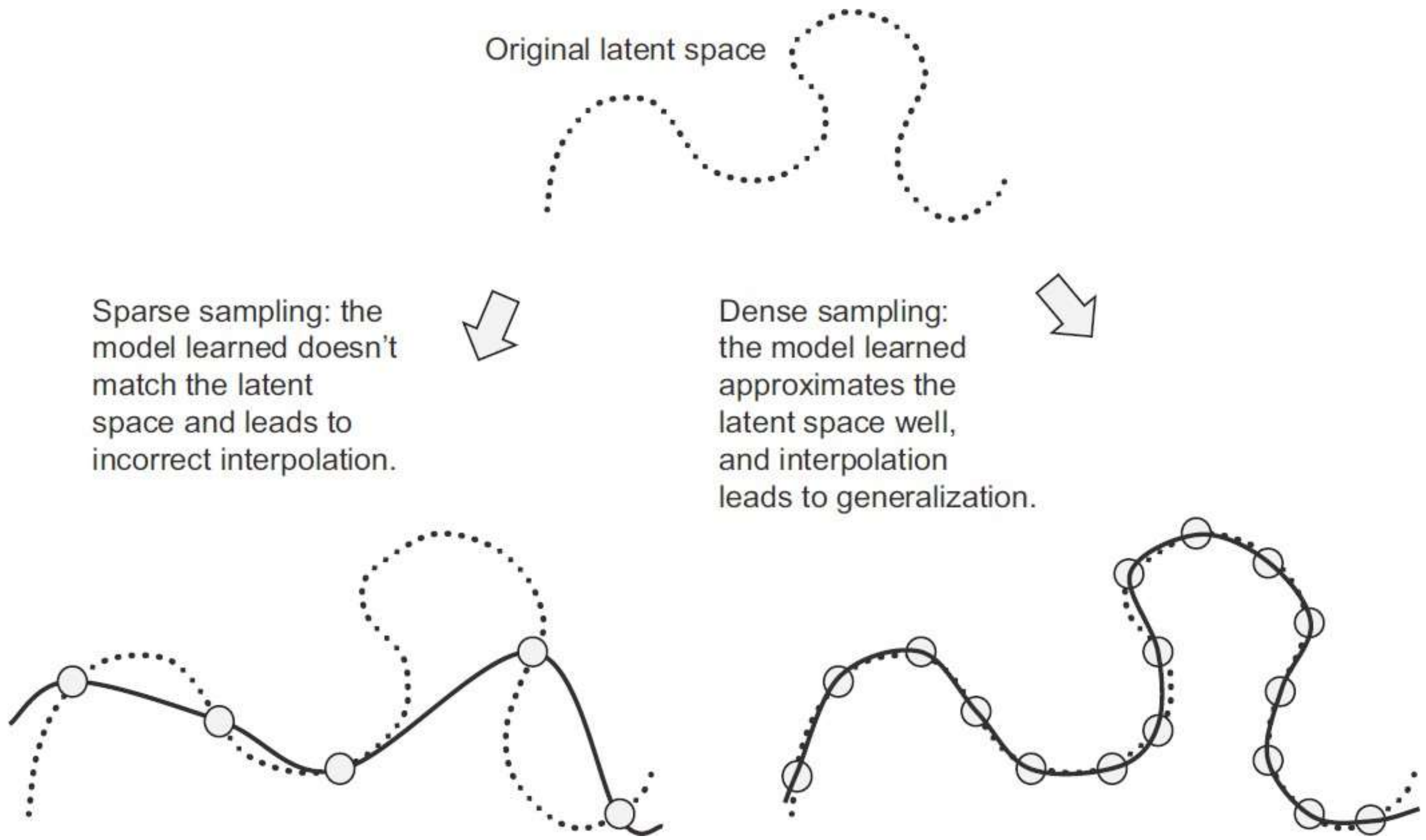
Training Data is Paramount

- While deep learning is indeed well suited to manifold learning, the power to generalize is more a consequence of the natural structure of your data than a consequence of any property of your model.
- The more informative and the less noisy your features are, the better you will be able to generalize, since your input space will be simpler and better structured.

Training Data is Paramount

- Data curation and feature engineering are essential to generalization.
- Further, because deep learning 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.

Dense Sampling is Necessary


Original latent space
Sparse sampling: the model learned doesn't match the latent space and leads to incorrect interpolation.
Dense sampling: the model learned approximates the latent space well, and interpolation leads to generalization.

A dense sampling of the input space is necessary in order to learn a model capable of accurate generalization

Training Data is Paramount…

- The best way to improve a deep learning 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.

Training Data is Paramount…

- 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.*

Evaluating machine learning models

Evaluating machine learning models

- You can only control what you can observe.
- Since your 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.

.Training, validation, and test sets

- Evaluating a model always boils down to splitting the available data into three sets: training, validation, and test
- Set apart some fraction of your data as your test set.
- Train on the remaining data, and evaluate on the test set.

.Training, validation, and test sets

- Why not have only two sets: a training set and a test set?
  - 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
- In order to prevent information leaks, you shouldn’t tune your model based on the test set, and therefore you should *also* reserve a validation set.

.Information Leaks

- Every time you tune a hyperparameter of your model based on the model’s performance on the validation set, some information about the validation data leaks into the model.
- If you do this only once, for one parameter, then very few bits of information will leak, and your validation set will remain reliable for evaluating the model.
- If you repeat this many times, then you’ll leak an increasingly significant amount of information about the validation set into the model.

.Training, validation, and test sets

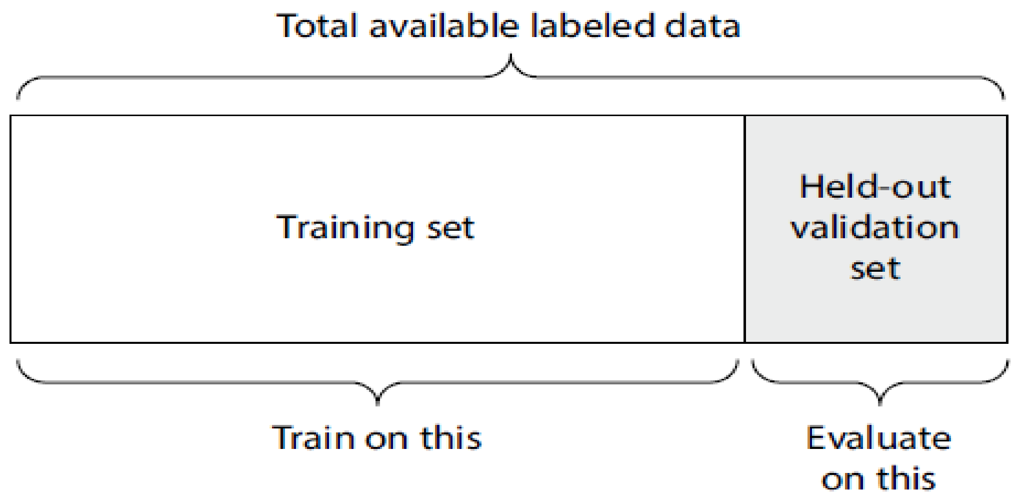
- At the end of the day, you'll end up with a model that performs artificially well on the validation data, because that's what you optimized it for.
- You care about performance on completely new data, not on the validation data, so you need to use a completely different, never-before-seen dataset to evaluate the model: the test dataset.
- Your 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 test set performance, then your measure of generalization will be flawed.

.Evaluation Protocols

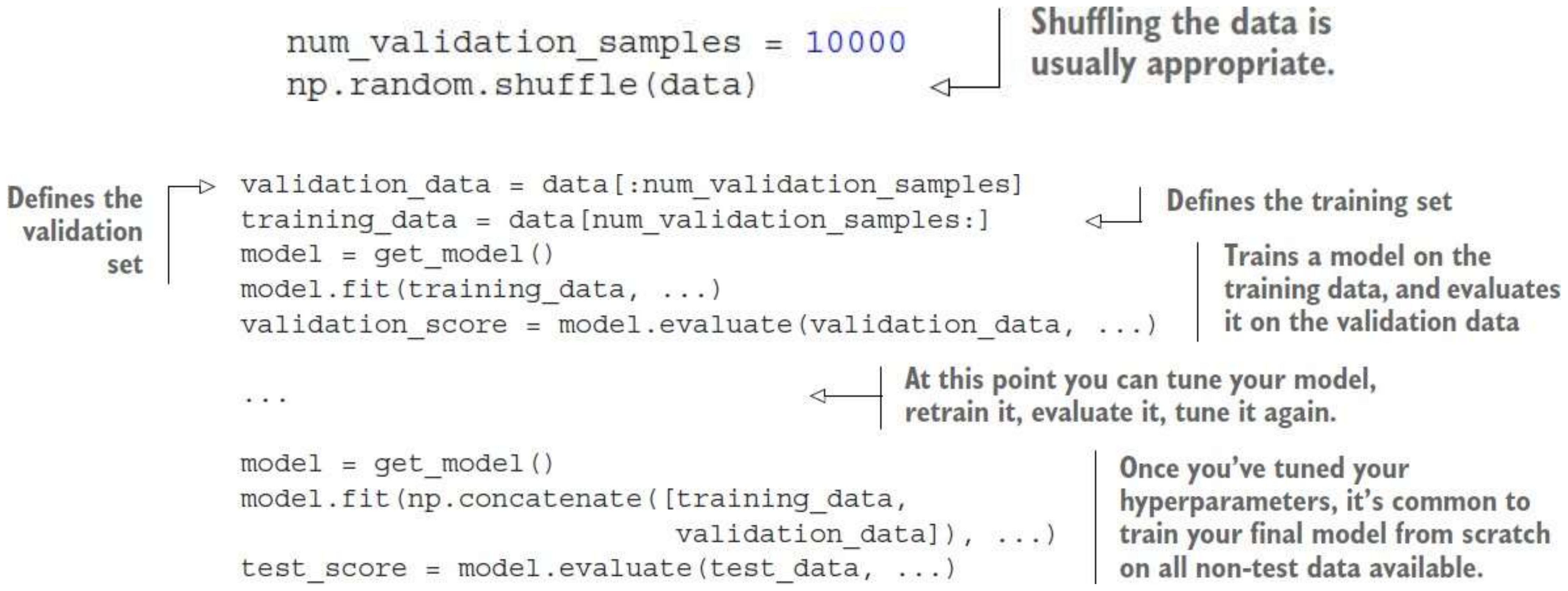
- Three evaluation protocols:
  - simple holdout validation
  - K-fold validation
  - iterated K-fold validation with shuffling

.Simple Holdout Validation

- Set apart some fraction of your data as your test set.
- Train on the remaining data, and evaluate on the test set.



.Simple Holdout 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, ...)
```

Shuffling the data is usually appropriate.

Defines the validation set

Defines the training set

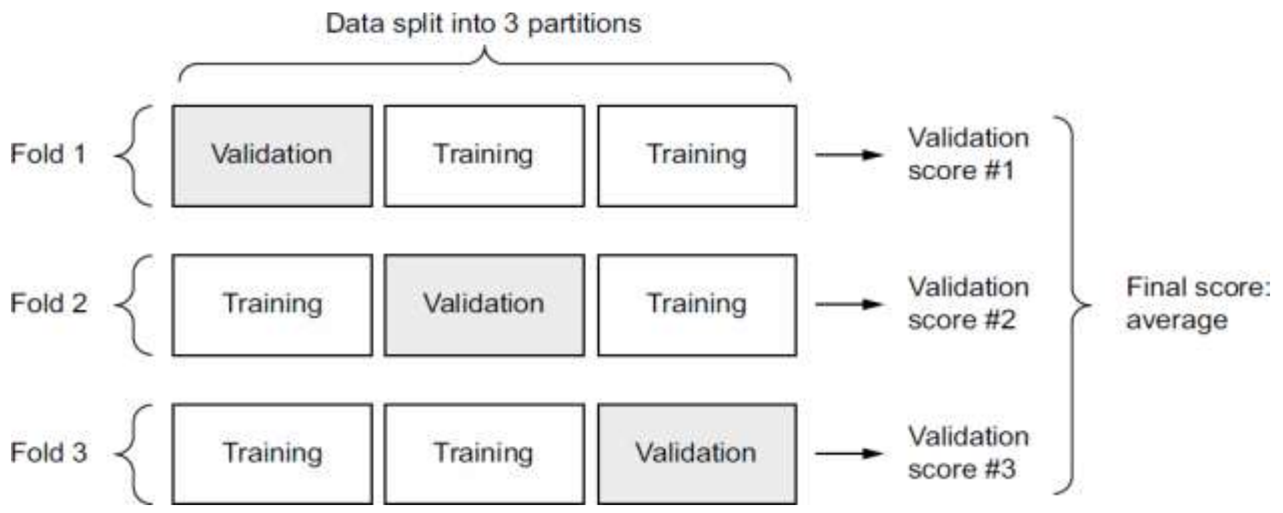
Trains a model on the training data, and evaluates it on the validation data

At this point you can tune your model, retrain it, evaluate it, tune it again.

Once you've tuned your hyperparameters, it's common to train your final model from scratch on all non-test data available.

.K-fold Validation...

- Split your 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.
- Your final score is then the averages of the K scores obtained.



- This method is helpful when the performance of your model shows significant variance based on your train-test split.

```
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, ...)
```

Selects the validation-data partition

Uses the remainder of the data as training data. Note that the + operator represents list concatenation, not summation.

Creates a brand-new instance of the model (untrained)

Validation score: average of the validation scores of the k folds

Trains the final model on all non-test data available

.Iterated K-fold Validation With Shuffling

- This one is for situations in which you have relatively little data available and you need to evaluate your model as precisely as possible.
- 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.
- Very expensive!!!

Beating a common-sense baseline

- Besides the different evaluation protocols, one last thing you should know about is the use of common-sense baselines.
- Before you start working with a dataset, you should always pick a trivial baseline that you'll try to beat.
- If you cross that threshold, your model is actually using the information in the input data to make predictions that generalize, and you can keep going.

Beating a common-sense baseline

- The baseline can be the performance of a random classifier or the performance of the simplest non machine learning technique.
  - in the MNIST digit-classification example, a simple baseline would be a validation accuracy greater than 0.1 (random classifier);
  - in the IMDB example, it would be a validation accuracy greater than 0.5.
  - In the Reuters example, it would be around 0.18-0.19, due to class imbalance.
  - In a binary classification problem where 90% of samples belong to class A and 10% to class B, then a classifier that always predicts A already achieves 0.9 in validation accuracy, and you'll need to do better than that.

Things to keep in mind about model evaluation

- *Data representativeness:* you usually should *randomly shuffle* your data before splitting it into training and test sets. [example digit classification]

Things to keep in mind about model evaluation

- *The arrow of time*—If you're trying to predict the future given the past (e.g. tomorrow's weather, stock movements), you should not randomly shuffle your data before splitting it, otherwise your model will effectively be trained on data from the future.
  - In such situations, you should always make sure all data in your test set is *posterior* to the data in the training set.

Things to keep in mind about model evaluation

- *Redundancy in your data*—If some data points in your data appear twice, then shuffling the data and splitting it into a training set and a validation set will result in redundancy.
  - Make sure your training set and validation set are disjoint.

Improving model fit

Improving model fit

- To achieve the perfect fit, you must first overfit.
- Since you don’t know in advance where the boundary lies, you must cross it to find it.

Improving model fit

- There are three common problems you'll encounter at this stage:
  - Training doesn't get started: your training loss doesn't go down over time.
  - Training gets started just fine, but your model doesn't meaningfully generalize: you can't beat the common-sense baseline you set.
  - Training and validation loss both go down over time, and you can beat your baseline, but you don't seem to be able to overfit, which indicates you're still underfitting.
- Your initial goal is getting a model that has some generalization power (it can beat a trivial baseline) and that is **able to overfit**.

Improving model fit

- There are three common techniques to address these issues and thus improving model fit:
  - Tuning key gradient descent parameters
  - Leveraging better architecture priors
  - Increasing model capacity

.Tuning key gradient descent parameters

- Sometimes training doesn’t get started, or it stalls too early. The loss is stuck.
- It’s always a problem with the configuration of the gradient descent process:
  - choice of optimizer
  - initial values of weights
  - learning rate
  - batch size

.Tuning key gradient descent parameters

- It is usually sufficient to tune the learning rate and the batch size while keeping the rest of the parameters constant.

Try:

- Lowering or increasing the learning rate
  - A learning rate that is too high may lead to updates that vastly overshoot a proper fit.
  - 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).

.Leveraging better architecture priors

- Your model trains but doesn’t generalize. What’s going on?
    - It indicates that *something is fundamentally wrong with the approach*

.Leveraging better architecture priors

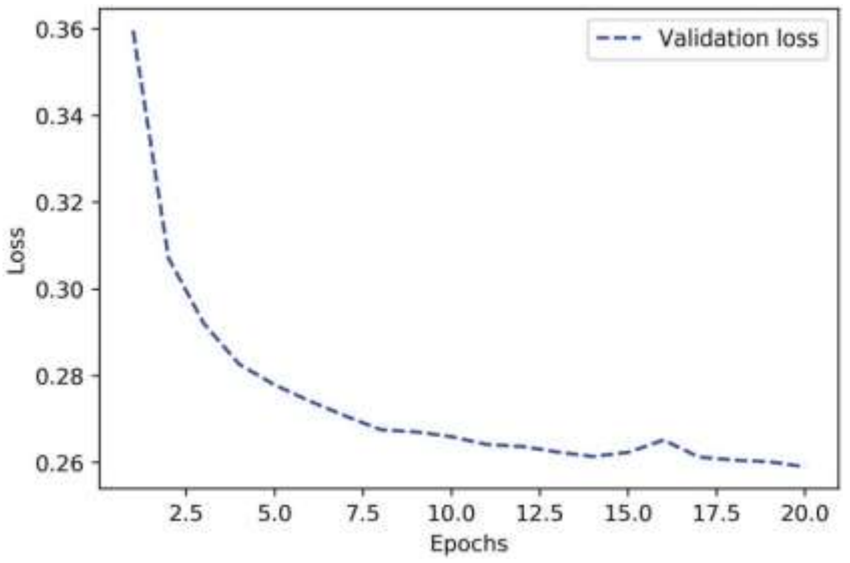
Some Tips:

- It may be the input data simply doesn’t contain sufficient information to predict the targets.
  - For example, we tried to fit an MNIST model where the labels were shuffled.
- It may be that the kind of model is not suited for the problem at hand
  - For example, we tried a densely connected architecture for a timeseries prediction, whereas a more appropriate recurrent architecture does manage to generalize well.

.Increasing model capacity

- Validation metrics seem to stall, or to improve very slowly, instead of peaking and reversing course.
- The validation loss goes to 0.26 and just stays there. You can fit, but you can't clearly overfit, even after many iterations over the training data.

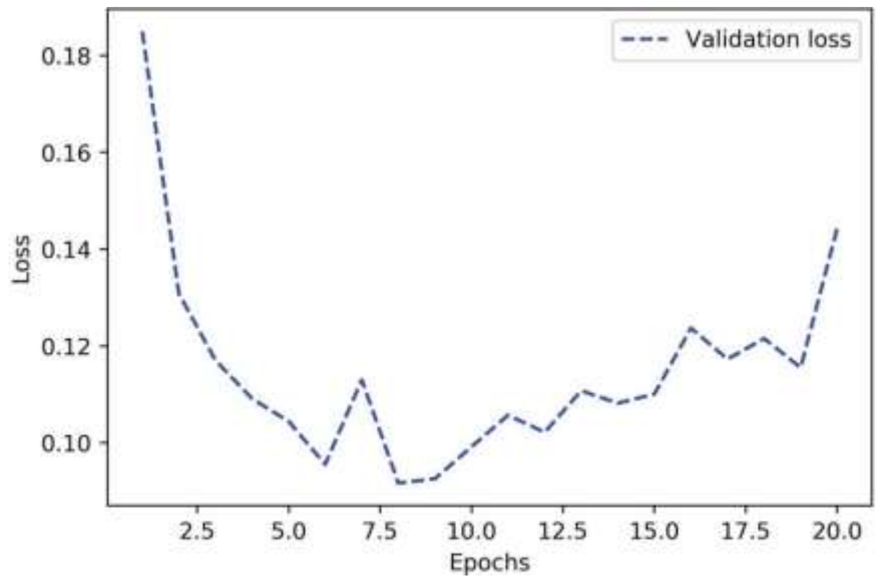
```
model = keras.Sequential([layers.Dense(10, activation="softmax")])
model.compile(optimizer="rmsprop",
              loss="sparse_categorical_crossentropy",
              metrics=["accuracy"])
history_small_model = model.fit(
    train_images, train_labels,
    epochs=20,
    batch_size=128,
    validation_split=0.2)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



.Increasing model capacity…

- If you can’t seem to be able to overfit, you’re going to need a bigger model, one with more capacity, able to store more information.
- You can increase representational power by adding more layers, using bigger layers (with more parameters), or using more appropriate layers for the problem at hand. …The model fits now fast and starts overfitting after 8 epochs

```
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)
```



Improving generalization

Improving generalization

- Once your model has shown itself to have some generalization power and to be able to overfit, it’s time to switch your focus to maximizing generalization.
- The following techniques can be used to maximizing generalization:
  - Dataset curation
  - Feature engineering
  - Using early stopping
  - Regularizing the model

.Dataset curation

- Generalization in deep learning originates from the latent structure of your data.
- If your data makes it possible to smoothly interpolate between samples, you will be able to train a deep learning model that generalizes.
- If your problem is overly noisy or fundamentally discrete, say, list sorting, deep learning will not help you.
- Deep learning is curve fitting, not magic.

.Dataset curation

- 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 you need a dense sampling of the input- cross-output space. More data will yield a better model.
  - Minimize labeling errors—visualize your inputs to check for anomalies, and proofread your labels.
  - Clean your data and deal with missing values.
  - If you have many features and you aren’t sure which ones are actually useful, do feature selection.

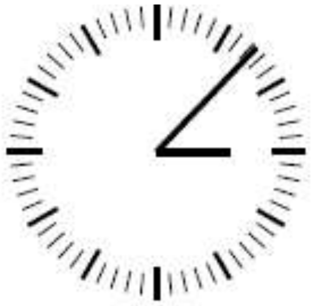
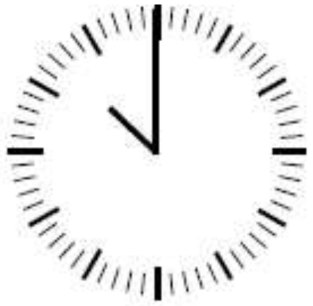
.Feature engineering

- Feature engineering is the process of using your own knowledge about the data and about the machine learning algorithm at hand to make the algorithm work better by applying hardcoded (non-learned) transformations to the data before it goes into the model.

.Feature engineering

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

Feature engineering for reading the time on a clock

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

.Using early stopping

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

.Using early stopping

- If training the models for longer than needed to figure out the number of epochs that yielded the best validation metrics, and then retrain a new model for exactly that number of epochs is expensive
- 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.

.Regularizing your 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 more “regular,” its curve smoother, more “generic”;
- It is less specific to the training set and better able to generalize by more closely approximating the latent manifold of the data.
- Regularizing a model is a process that should always be guided by an accurate evaluation procedure. You will only achieve generalization if you can measure it.

.Regularizing your model

- The most common regularization techniques are:
  - Reducing the network’s size
  - Adding weight regularization
  - Adding dropout

.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 layers and the number of units per layer).
- At the same time, models need to have enough parameters that they don’t underfit: your model shouldn’t be starved for memorization resources.
- There is a compromise to be found between too much capacity and not enough capacity.

.Reducing The Network’s Size

- Unfortunately, there is no magical formula to determine the right number of layers or the right size for each layer.
- You must evaluate an array of different architectures (on your validation set) in order to find the correct model size for your 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 you see diminishing returns with regard to validation loss.

Original Model

```
model = keras.Sequential([
    layers.Dense(16, activation="relu"),
    layers.Dense(16, activation="relu"),
    layers.Dense(1, activation="sigmoid")
])
```

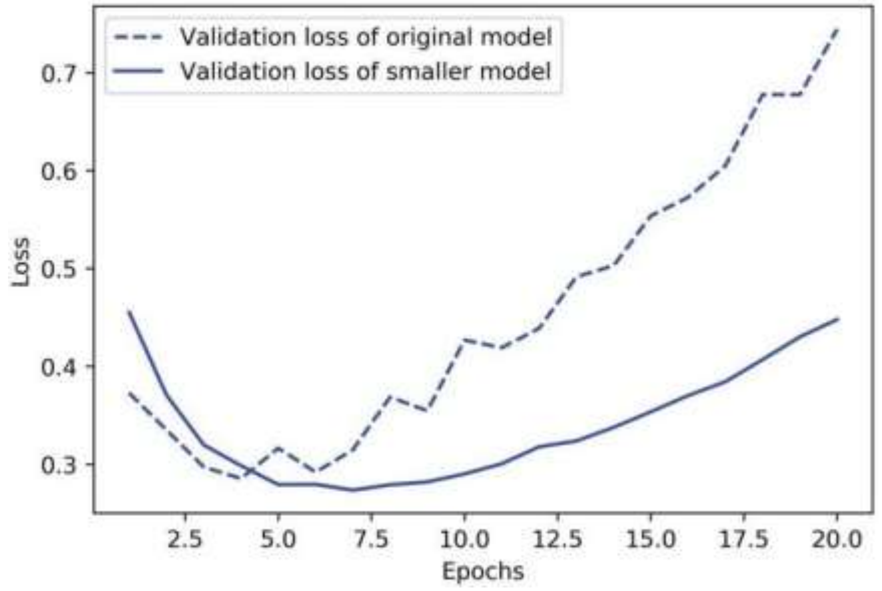
Smaller Model

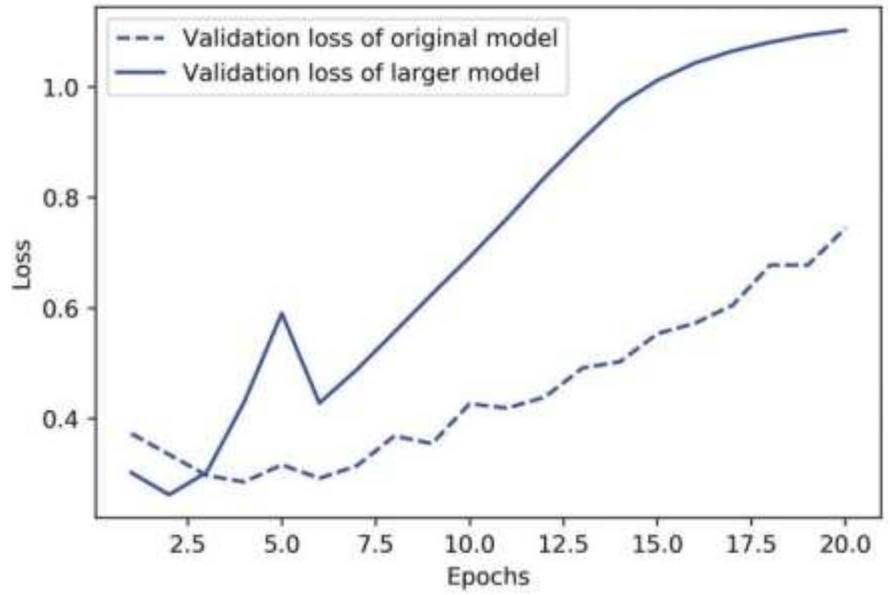
```
model = keras.Sequential([
    layers.Dense(4, activation="relu"),
    layers.Dense(4, activation="relu"),
    layers.Dense(1, activation="sigmoid")
])
```

Larger Model

```
model = keras.Sequential([
    layers.Dense(512, activation="relu"),
    layers.Dense(512, activation="relu"),
    layers.Dense(1, activation="sigmoid")
])
```

Original vs. Smaller Model





Original vs. Larger Model

.Reducing The Network's Size...

- 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.
- The bigger model starts overfitting almost immediately, after just one epoch, 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)

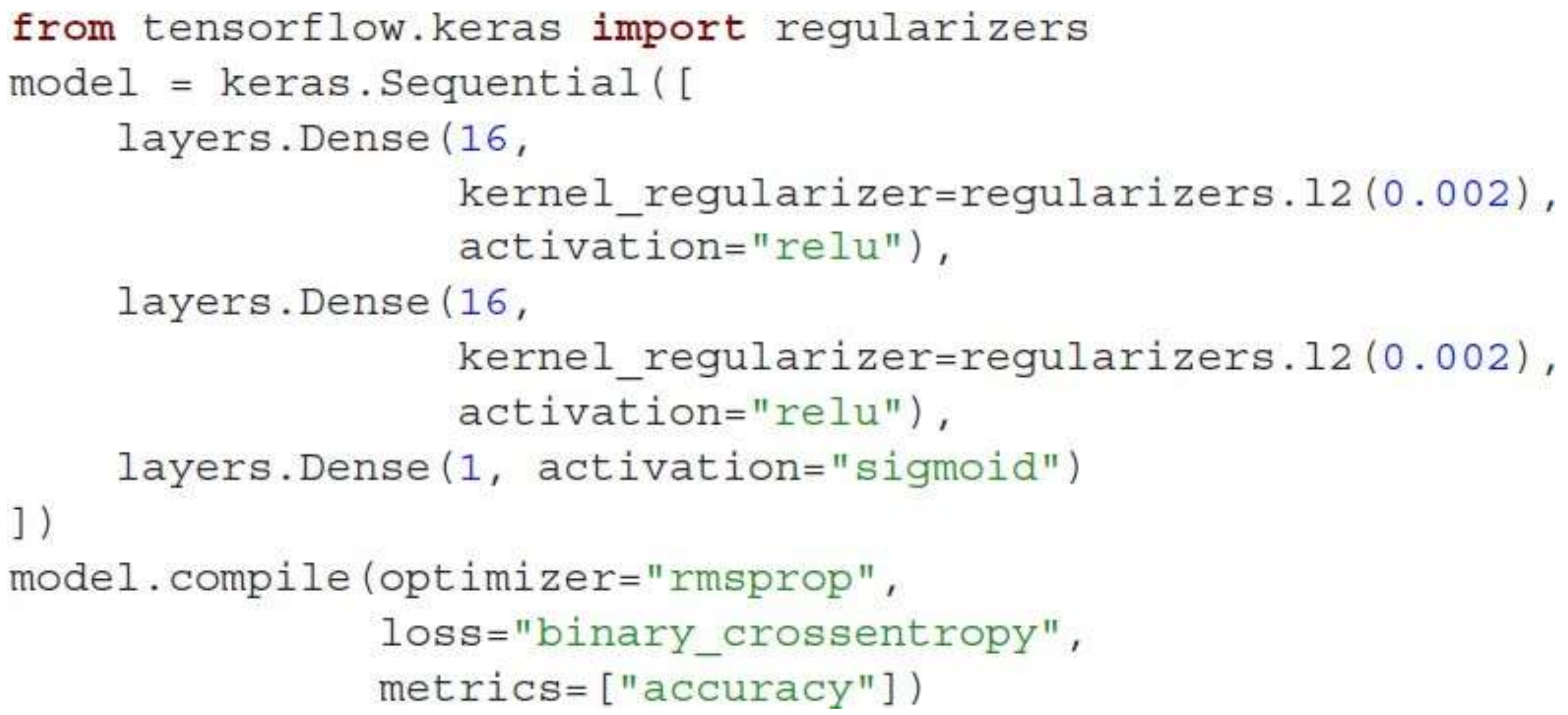
.Adding Weight Regularization

- 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*.

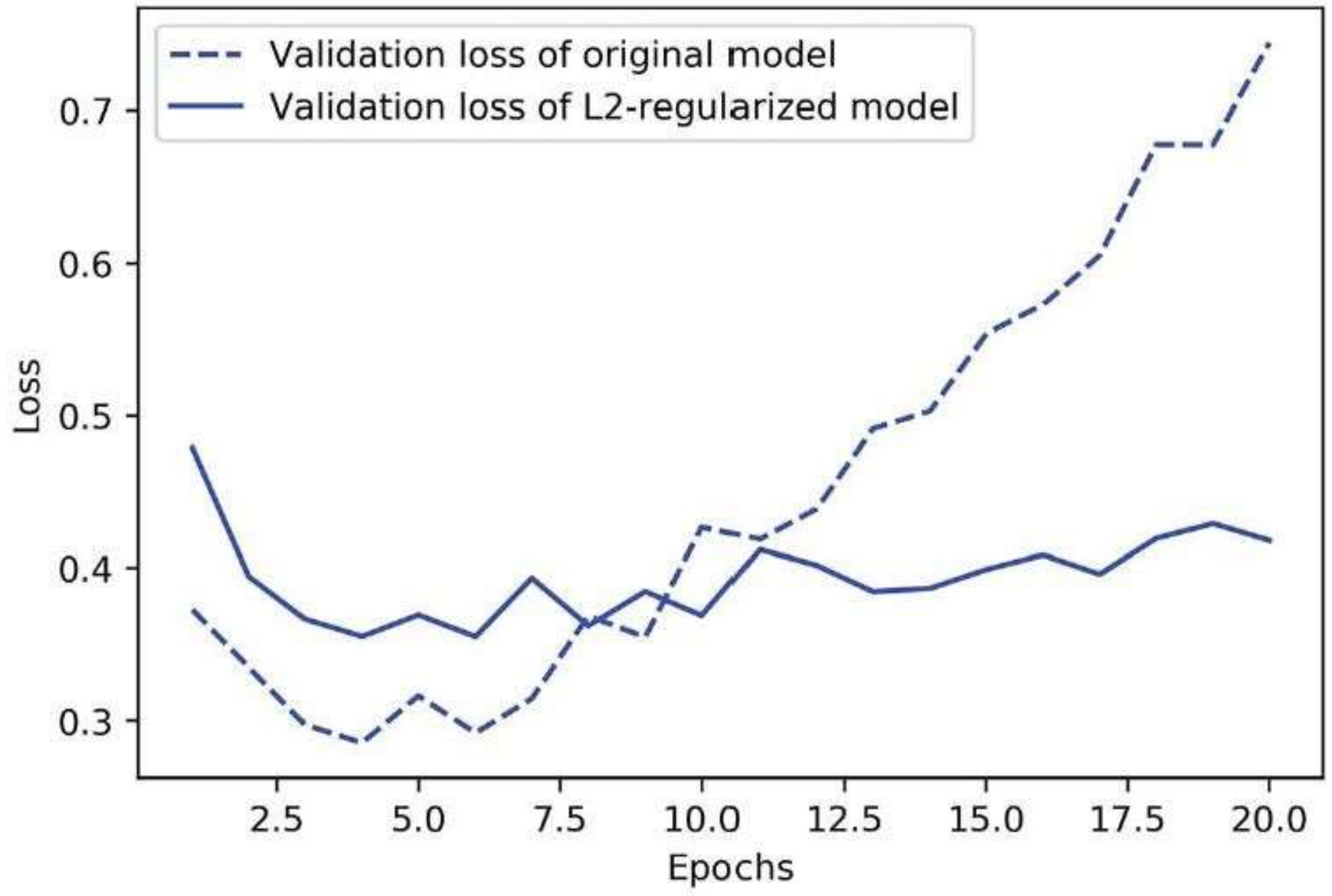
.Adding Weight Regularization

- It’s done by adding to the loss function of the model a cost associated with having large weights.
  - 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.

Adding L2 Regularizer in Keras



```
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"])
```

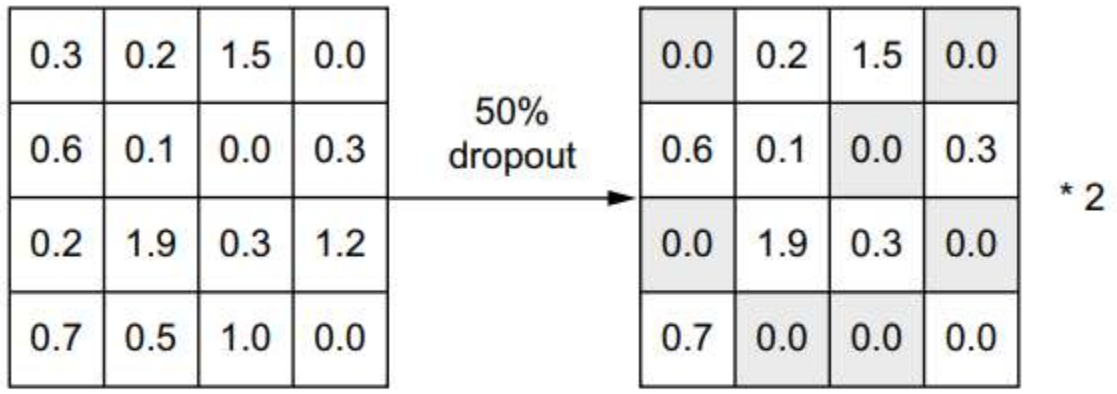

Validation loss of original model
Validation loss of L2-regularized model
0.7
0.6
0.5
0.4
0.3
Loss
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0
Epochs

.Adding Weight Regularization…

- Weight regularization is more typically used for smaller deep learning models.
- Large deep learning models tend to be so overparameterized that imposing constraints on weight values hasn’t much impact on model capacity and generalization.
- In these cases, a different regularization technique is preferred: dropout.

Adding Dropout

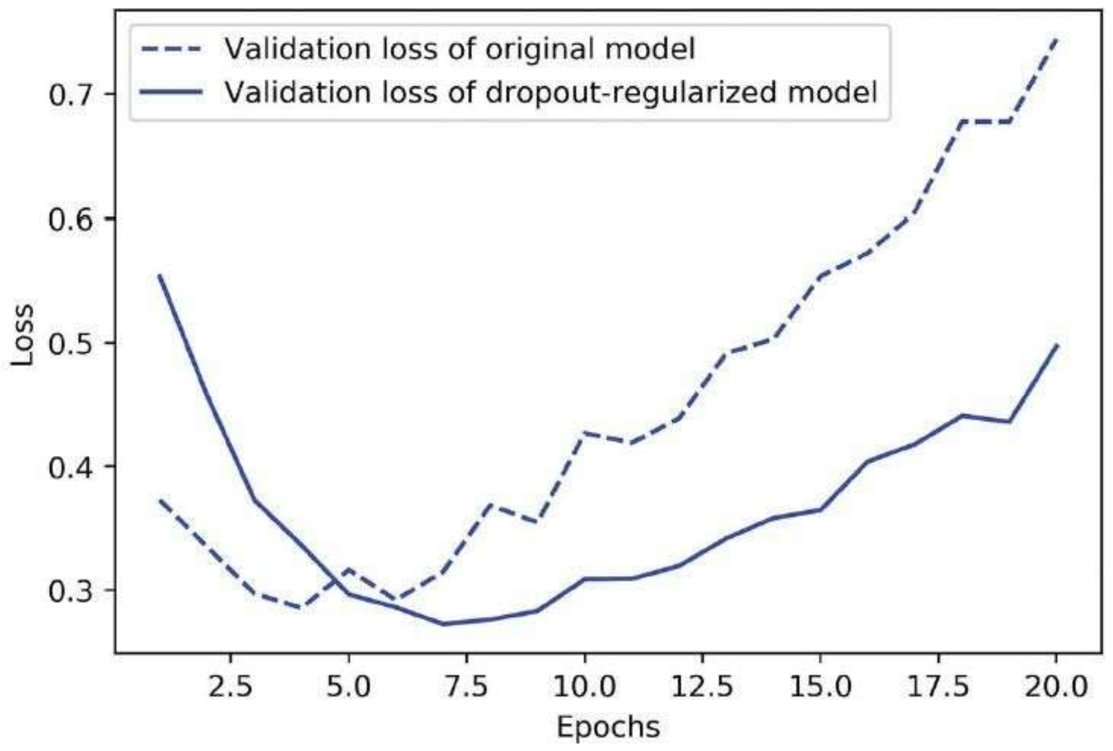
- Dropout is one of the most effective and most commonly used regularization techniques for neural networks;
- Dropout, applied to a layer, consists of randomly dropping out (setting to zero) a number of output features of the layer during training.



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

Adding dropout to the IMDB model

```
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)
```


Validation loss of original model
Validation loss of dropout-regularized model
Loss
Epochs