

CH5. Fundamentals of Machine Learning

Il-Youp Kwak
Chung-Ang University



Fundamentals of Machine Learning

Generalization: The goal of machine learning

Evaluating machine-learning models

Improving model fit

Improving generalization



Generalization: The goal of machine learning

The fundamental issue in machine learning is the tension between optimization and generalization



Figure 5.1 Canonical overfitting behavior

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

The processing of fighting overfitting is called **regularization**



Underfitting and overfitting

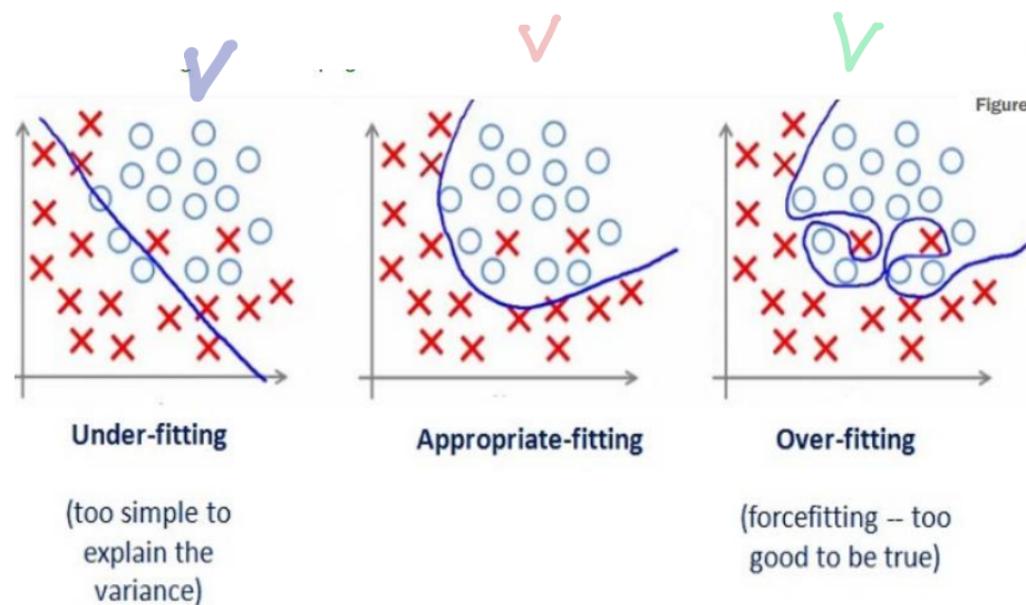


Figure 5.1 Canonical overfitting behavior



Ch5. Fundamentals of Machine Learning / Generalization: The goal of machine learning

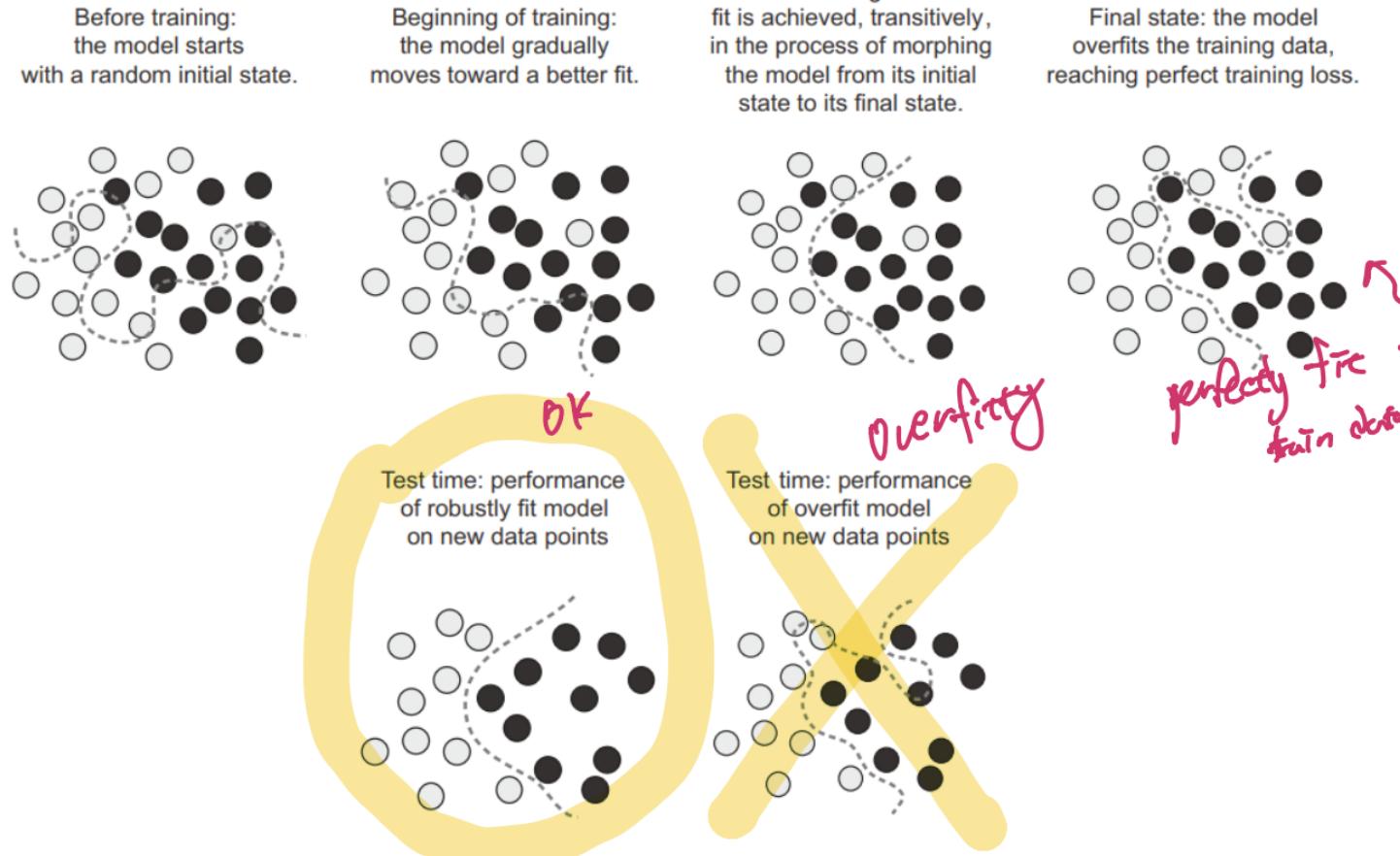


Figure 5.10 Going from a random model to an overfit model, and achieving a robust fit as an intermediate state



Noisy training data

In real-world datasets, it's fairly common for some inputs to be invalid.

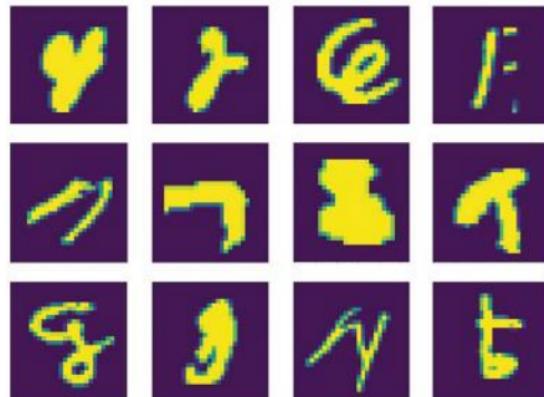


Figure 5.2 Some pretty weird
MNIST training samples

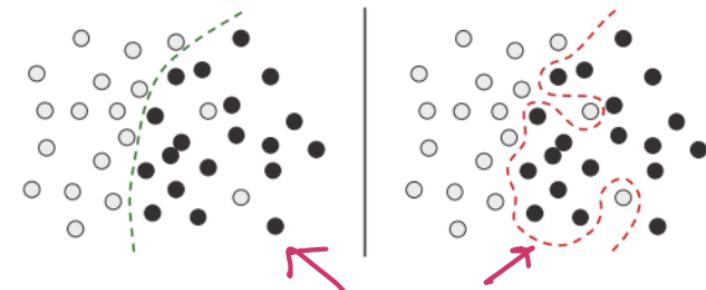


Figure 5.4 Dealing with outliers: robust fit vs. overfitting

robust model vs overfitting!!

Even worst:

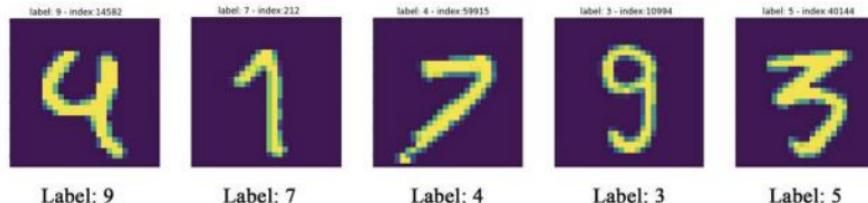


Figure 5.3 Mislabeled MNIST training samples



Rare features and suspicious correlations

Datasets that include rare feature values are highly susceptible to overfitting

Example of suspicious correlations:

Listing 5.1 Adding white noise channels or all-zeros channels to MNIST

```
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)
```

→ $(60000, 1568)^{\text{sf}}$

→ $(60000, 1568)^{\text{sf}}$



Ch5. Fundamentals of Machine Learning / Generalization: The goal of machine learning

Listing 5.2 Training the same model on MNIST data with noise channels or all-zero channels

```
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)

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



Listing 5.3 Plotting a validation accuracy comparison

```
import matplotlib.pyplot as plt
val_acc_noise = history_noise.history["val_accuracy"]
val_acc_zeros = history_zeros.history["val_accuracy"]
epochs = range(1, 11)
plt.plot(epochs, val_acc_noise, "b-",
         label="Validation accuracy with noise channels")
plt.plot(epochs, val_acc_zeros, "b--",
         label="Validation accuracy with zeros channels")
plt.title("Effect of noise channels on validation accuracy")
plt.xlabel("Epochs")
plt.ylabel("Accuracy")
plt.legend()
```

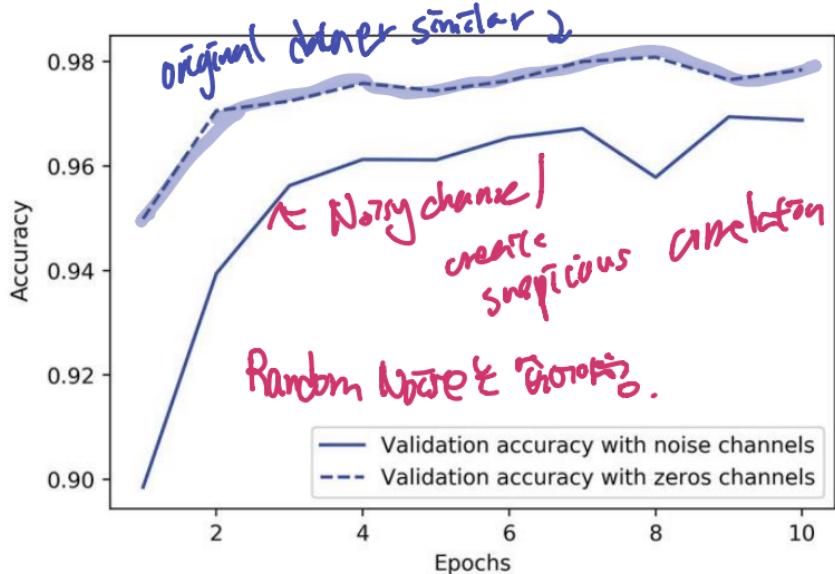
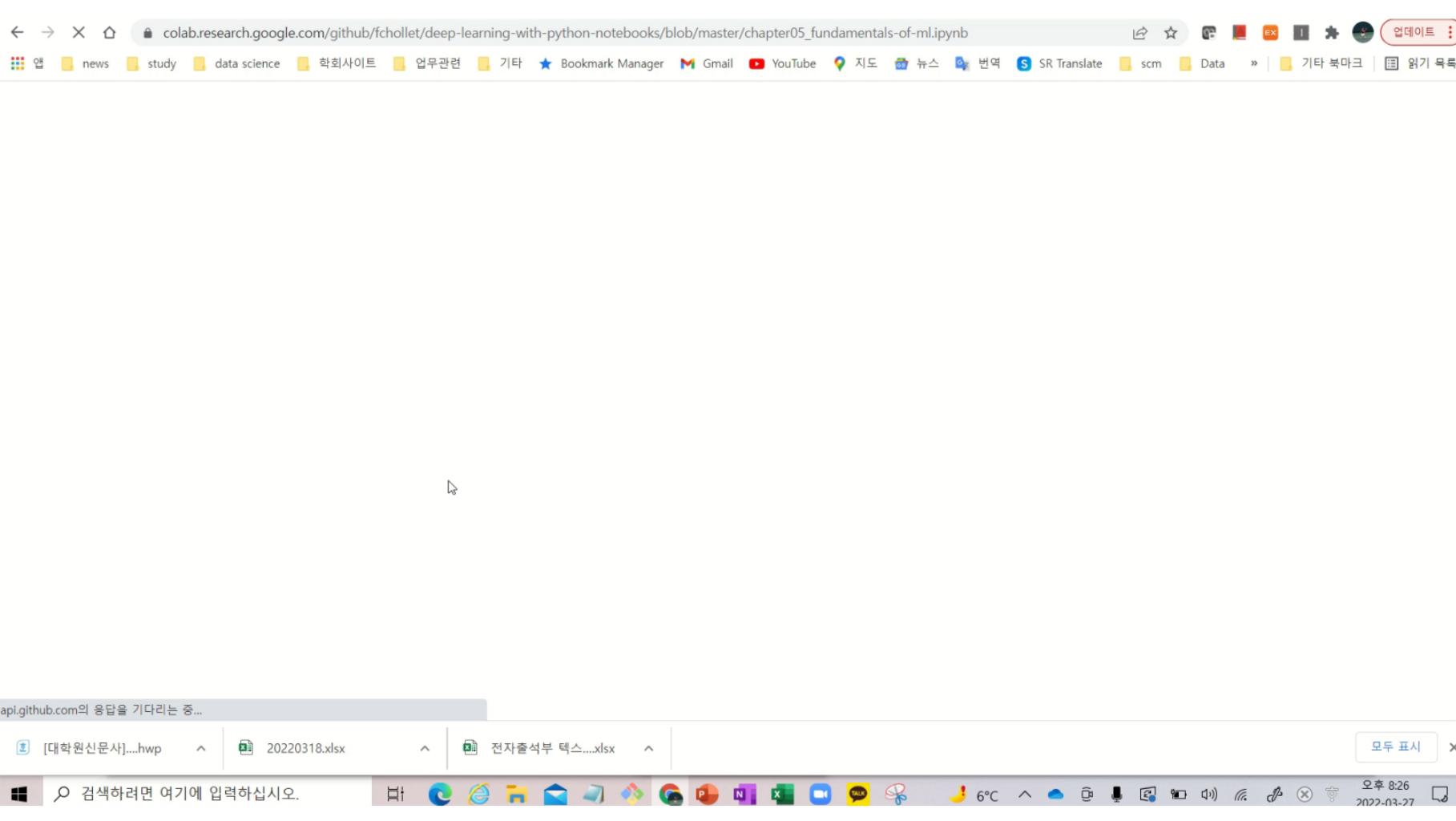


Figure 5.6 Effect of noise channels on validation accuracy

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.





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

Listing 5.4 Fitting an MNIST model with randomly shuffled labels

```
(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"])
model.fit(train_images, random_train_labels,
          epochs=100,
          batch_size=128,
          validation_split=0.2)
```

Even though there is no relationship whatsoever between the inputs and the shuffled labels, the training loss goes down just fine

(Accuracy...)

Naturally, the validation loss does not improve at all over time



- ▼ The nature of generalization in deep learning

Fitting a MNIST model with randomly shuffled labels

```
[ ] [(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"])
model.fit(train_images, random_train_labels,
          epochs=100,
          batch_size=128,
          validation_split=0.2)
```

The manifold hypothesis

The manifold hypothesis

Actual handwritten digits only occupy a tiny subspace of the parent space of all possible 28×28 uint8 arrays



All samples in the valid subspace are connected by smooth paths that run through the subspace.

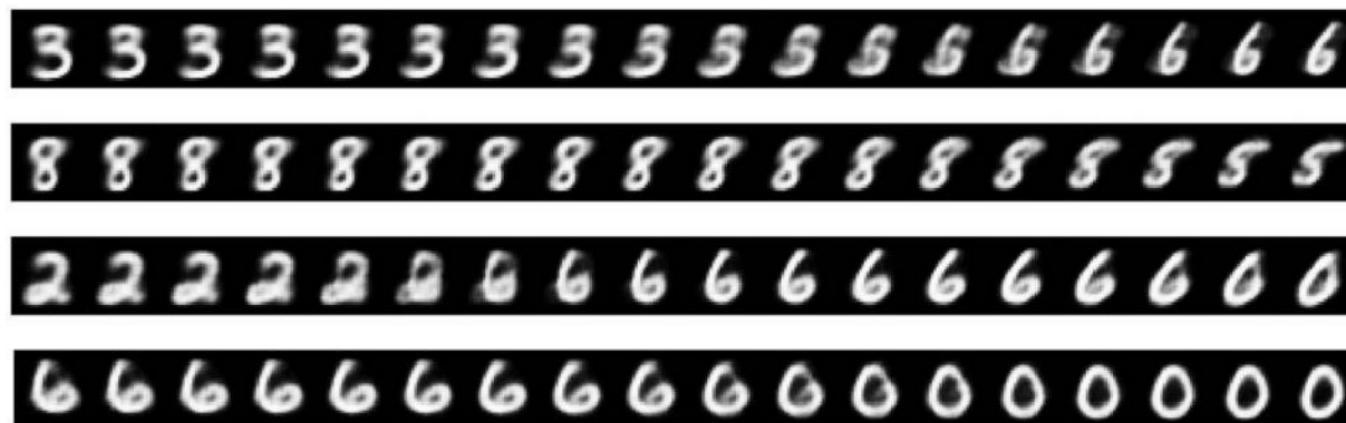


Figure 5.7 Different MNIST digits gradually morphing into one another, showing that the space of handwritten digits forms a “manifold.” This image was generated using code from chapter 12.

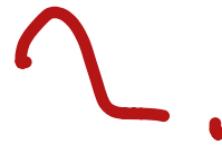


The manifold hypothesis

Manifold hypothesis posits that all natural data lies on a low-dimensional manifold within the high-dimensional space where it is encoded

Machine learning models only have to fit relatively simple, low-dimensional, highly structured subspaces within their potential input space (latent manifolds).

Within one of these manifolds, it's always possible to interpolate between two inputs, that is to say, morph one into another via a continuous path along which all points fall on the manifold.



Interpolation as a source of generalization

Deep learning achieves generalization via interpolation on a learned approximation of the data manifold

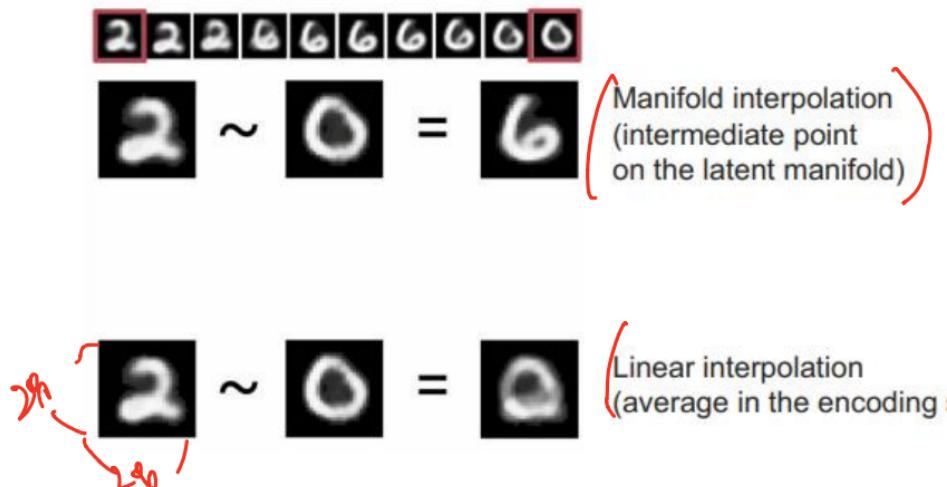


Figure 5.8 Difference between linear interpolation and interpolation on the latent manifold. Every point on the latent manifold of digits is a valid digit, but the average of two digits usually isn't.



Why deep learning works?

A sheet of paper represents a 2D manifold within 3D space. A deep learning model is a tool for uncrumpling paper balls, that is, for disentangling latent manifolds.



Figure 5.9 Uncrumpling a complicated manifold of data



Training data is paramount

Deep learning is curve fitting, for a model to perform well it needs to be trained on a dense sampling of its input space

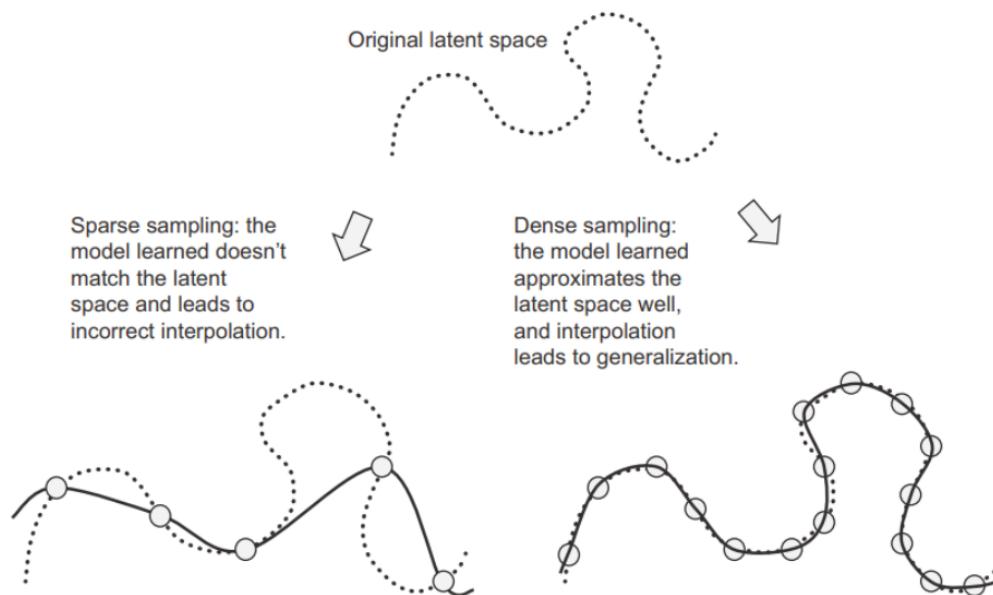


Figure 5.11 A dense sampling of the input space is necessary in order to learn a model capable of accurate generalization.



Evaluating machine-learning models

Goal is to achieve models that generalize

It's essential to be able to reliably measure the generalization power of your model



Training, validation, and test sets

Tuning **parameters** and **hyperparameters** is a form of **learning**

Tuning is based on its performance on validation set

Tuning a lot would cause **information leak**

Thus, you need never-before-seen test dataset



Simple hold-out validation

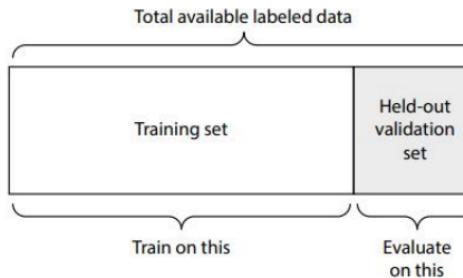


Figure 5.12 Simple holdout validation split

Listing 5.5 Holdout validation (note that labels are omitted for simplicity)

```

num_validation_samples = 10000           | Shuffling the data is
np.random.shuffle(data)                | usually appropriate.

Defines the validation set             |→ validation_data = data[:num_validation_samples]
                                         |→ training_data = data[num_validation_samples:]    | Defines the training set
model = get_model()                   |→ model.fit(training_data, ...)
model.fit(training_data, ...)         |→ validation_score = model.evaluate(validation_data, ...) | Trains a model on the
validation_score = model.evaluate(validation_data, ...) | training data, and evaluates it on the validation data

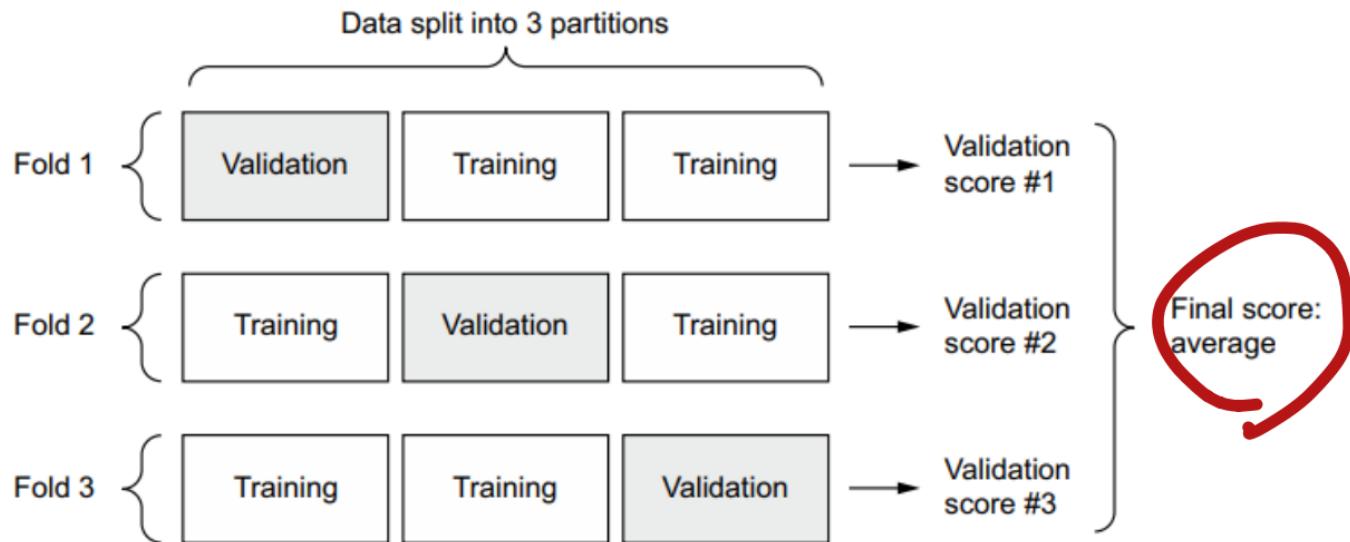
...                                     | At this point you can tune your model,
                                         | retrain it, evaluate it, tune it again.

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

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

Figure 5.13 K-fold cross-validation with $K=3$ 

K fold validation

Listing 5.6 K-fold cross-validation (note that labels are omitted for simplicity)

```
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

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

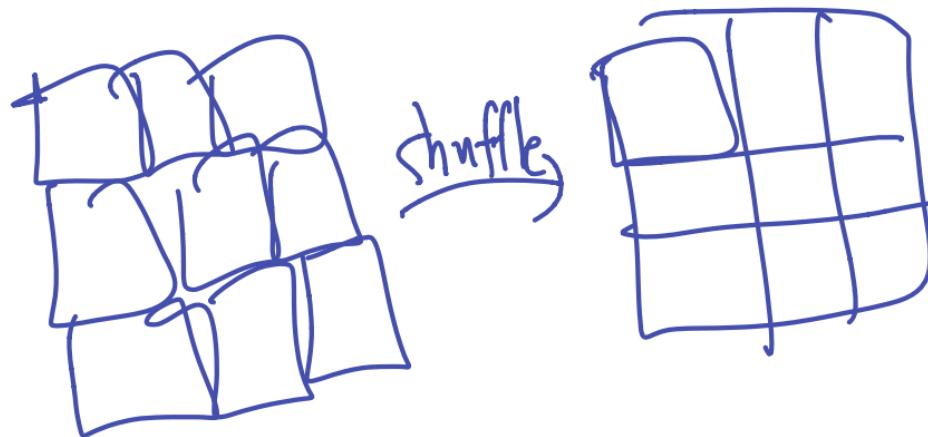
Uses the remainder of the data as training data. Note that the + operator represents list concatenation, not summation.



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

Applying K-fold validation multiple times, shuffling the data every time before splitting it K ways



Things to keep in mind

Data representativeness

Randomly shuffle data before splitting it

The arrow of time

If you are predicting future event, don't shuffle. Use future data as test set

Redundancy in your data

Remove redundant data before the start



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.

Once you have such a model, you'll focus on refining generalization by fighting overfitting.

Three common problems: 1) training loss doesn't go down, 2) model doesn't meaningfully generalize, 3) training and validation loss both go down but model still underfitting.



Tuning key gradient descent parameters

Sometimes training doesn't get started

When this happens, it's always a problem with the configuration of the gradient descent process

Listing 5.7 Training an MNIST model with an incorrectly high learning rate

```
(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"])
model.fit(train_images, train_labels,
          epochs=10,
          batch_size=128,
          validation_split=0.2)
```



Listing 5.8 The same model with a more appropriate learning rate

```
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"])
model.fit(train_images, train_labels,
          epochs=10,
          batch_size=128,
          validation_split=0.2)
```




Leveraging better architecture priors

Model trains but doesn't generalize. What's going on?

The kind of model you're using is not suited for the problem

In the following chapters, you'll learn about the best architectures to use for a variety of data modalities—images, text, timeseries, and so on.



Increasing model capacity

If you manage to get to a model that fits, you need to get your model to start overfitting.

Listing 5.9 A simple logistic regression on MNIST

```
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)
```

it's likely a problem with the representational power of your model

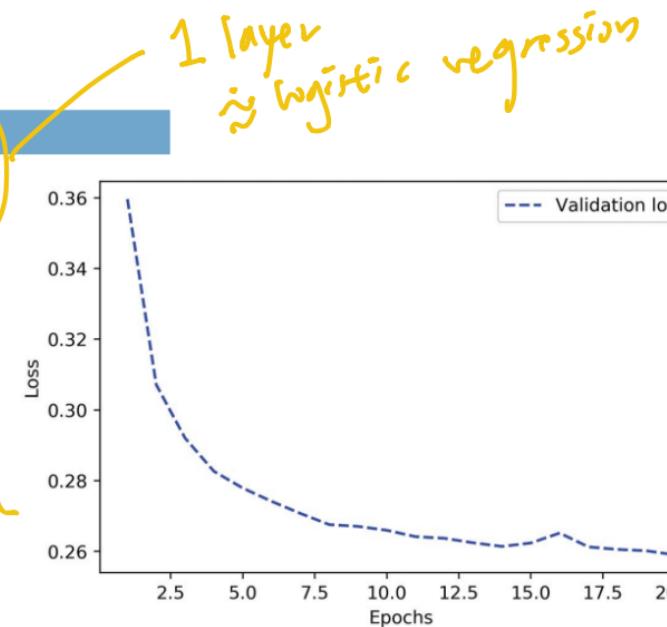


Figure 5.14 Effect of insufficient model capacity on loss curves



Use bigger model

```
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)
```

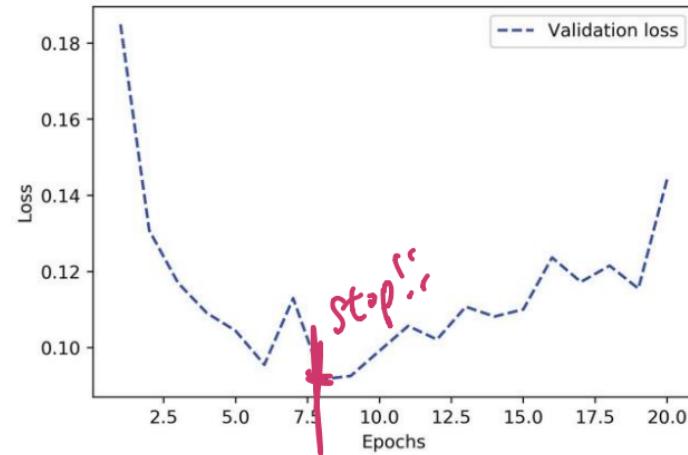
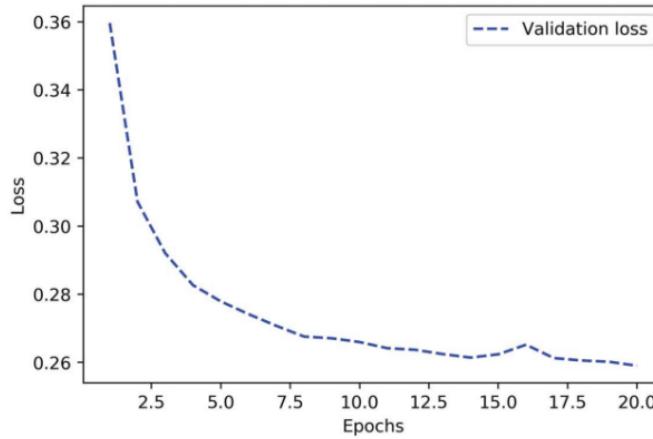


Figure 5.14 Effect of insufficient model capacity on loss curves

Figure 5.15 Validation loss for a model with appropriate capacity



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.

Dataset curation: Make sure you have enough data, Minimize labeling errors, Clean your data and deal with missing values, Feature selection, etc



Feature engineering

Feature engineering is the process of using your own knowledge about the data to make the algorithm work better by applying hardcoded (non-learned) transformations to the data

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

Good features allow you to solve problems using fewer resources

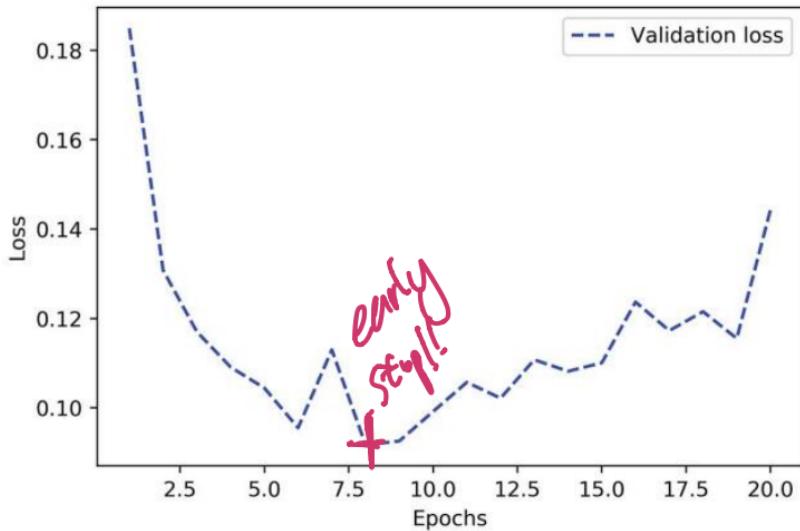
Good features let you solve a problem with far less data

Figure 5.16 Feature engineering for reading the time on a clock



Using early stopping

In deep learning, we always use models that are vastly overparameterized. Interrupt training long before you've reached the minimum possible training loss.



In Keras, it's typical to do this with an **EarlyStopping** callback

Figure 5.15 Validation loss for a model with appropriate capacity



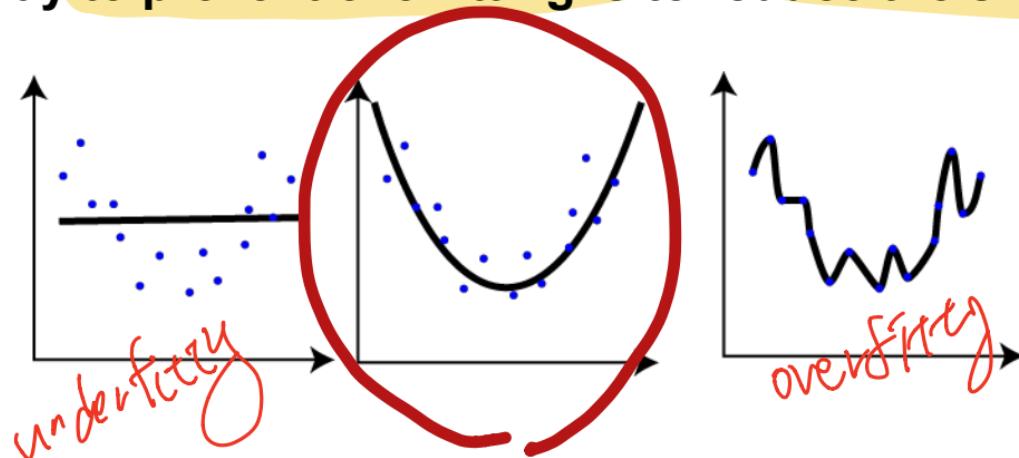
Regularizing your model

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.



Reducing the network's size

The simplest way to prevent overfitting is to reduce the size of the model



Reducing the network's size

Listing 5.10 Original model

```
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")
])
```

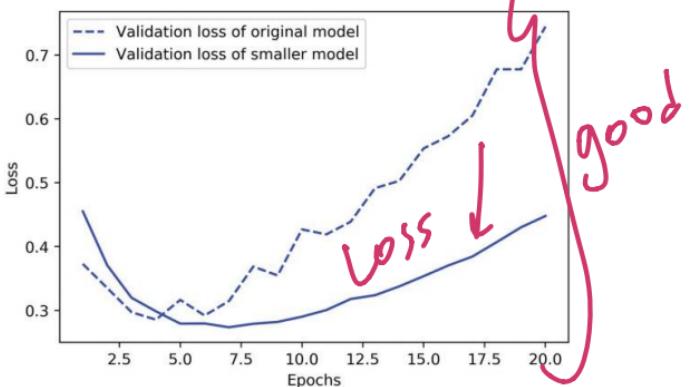


Figure 5.17 Original model vs. smaller model on IMDB review classification

Listing 5.11 Version of the model with lower capacity

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

Listing 5.12 Version of the model with higher capacity

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

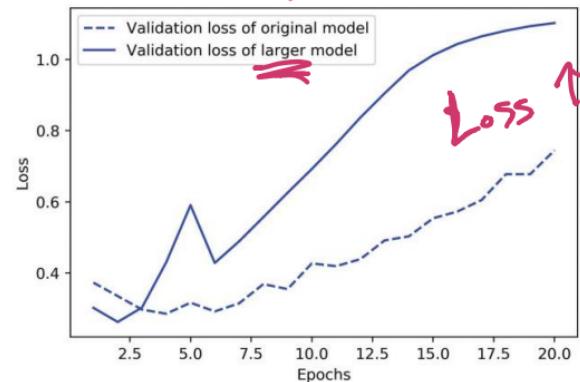


Figure 5.18 Original model vs. much larger model on IMDB review classification



Adding weight regularization

Simpler models are less likely to overfit than complex ones

Simple model is a model where the parameter values has less entropy

L1 regularization: Lasso *penalty*

$$C(\mathbf{w}, b) = \sum_{i=1}^m L(\hat{y}_i, y_i) + \frac{\lambda}{2m} \|\mathbf{w}\|$$

$$\|\mathbf{w}\| = \sum_{j=1}^{n_x} |w_j|$$

L2 regularization: Ridge *penalty*

$$C(\mathbf{w}, b) = \sum_{i=1}^m L(\hat{y}_i, y_i) + \frac{\lambda}{2m} \|\mathbf{w}\|^2$$

$$\|\mathbf{w}\|^2 = \sum_{j=1}^{n_x} w_j^2$$



Adding weight regularization

Listing 5.13 Adding L2 weight regularization to the model

```
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")
])
```

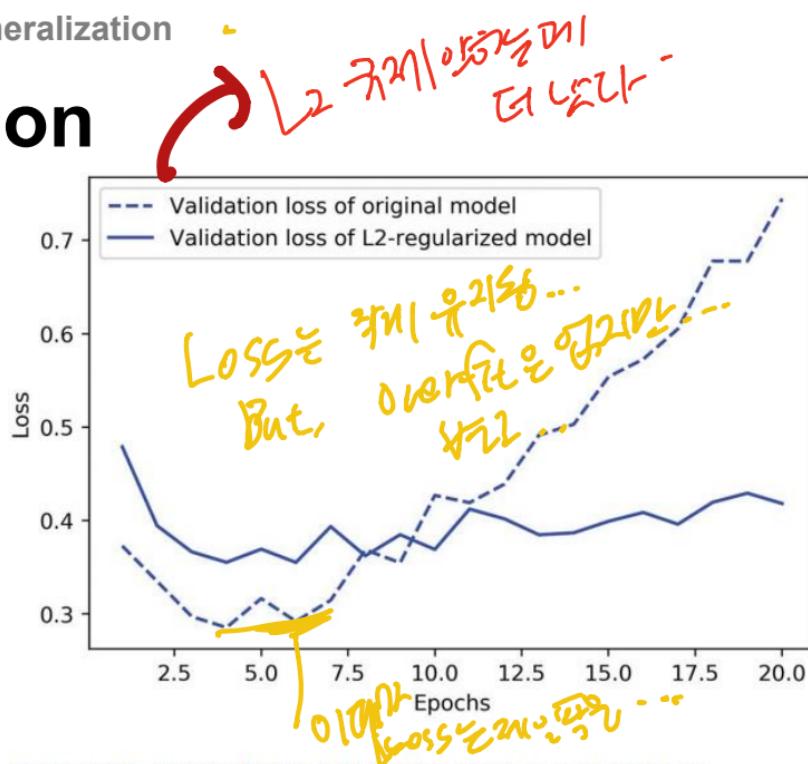


Figure 5.19 Effect of L2 weight regularization on validation loss

Listing 5.14 Different weight regularizers available in Keras

```
from tensorflow.keras import regularizers
regularizers.l1(0.001)           ← L1 regularization
regularizers.l1_l2(l1=0.001, l2=0.001) ← Simultaneous L1 and L2 regularization
```



Adding dropout

Randomly ***dropping out*** (setting to zero) a number of output features of the layer during training

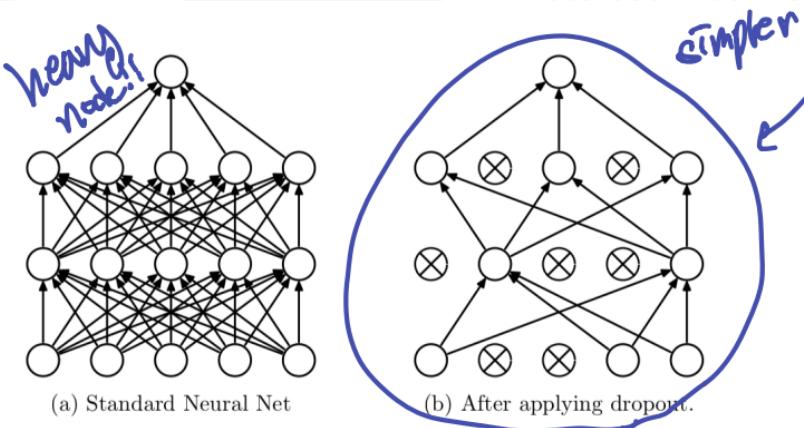
0.3	0.2	1.5	0.0
0.6	0.1	0.0	0.3
0.2	1.9	0.3	1.2
0.7	0.5	1.0	0.0

50% dropout →

0.0	0.2	1.5	0.0
0.6	0.1	0.0	0.3
0.0	1.9	0.3	0.0
0.7	0.0	0.0	0.0

* 2

Figure 5.20 Dropout applied to an activation matrix at training time, with rescaling happening during training. At test time the activation matrix is unchanged.



Adding dropout

Listing 5.15 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")
])
```

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

At testing time

`layer_output *= 0.5` ← At test time

Or, at training time

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

Note that we're scaling up rather
scaling down in this case.

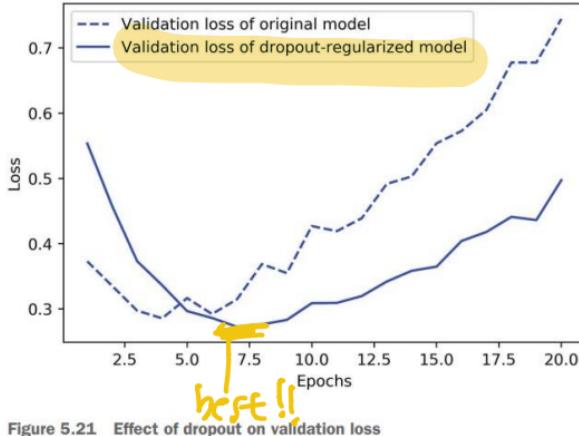


Figure 5.21 Effect of dropout on validation loss

At training time, drops out 50%
of the units in the output



Table of contents

- Training, validation, and test sets
- Simple hold-out validation
- K-fold validation
- Iterated K-fold validation with shuffling
- Beating a common-sense baseline
- Things to keep in mind about model evaluation
- Improving model fit
 - Tuning key gradient descent parameters**
 - Leveraging better architecture priors
 - Increasing model capacity
- Improving generalization
 - Dataset curation
 - Feature engineering
 - Using early stopping
 - Regularizing your model
 - Reducing the network's size
 - Adding weight regularization

+ Code + Text | ⌂ Copy to Drive

Things to keep in mind about model evaluation

▼ Improving model fit

▼ Tuning key gradient descent parameters

Training a MNIST model with an incorrectly high learning rate

```
(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"])
model.fit(train_images, train_labels,
          epochs=10,
          batch_size=128,
          validation_split=0.2)
```

Epoch 1/10

! 7s completed at 9:37 PM

Thank you!

