Deep Learning

Mohammad Reza Mohammadi

2021

How deep learning works?

- Optimizers (Gradient-based)
  - Backpropagation
- Loss and Activation functions

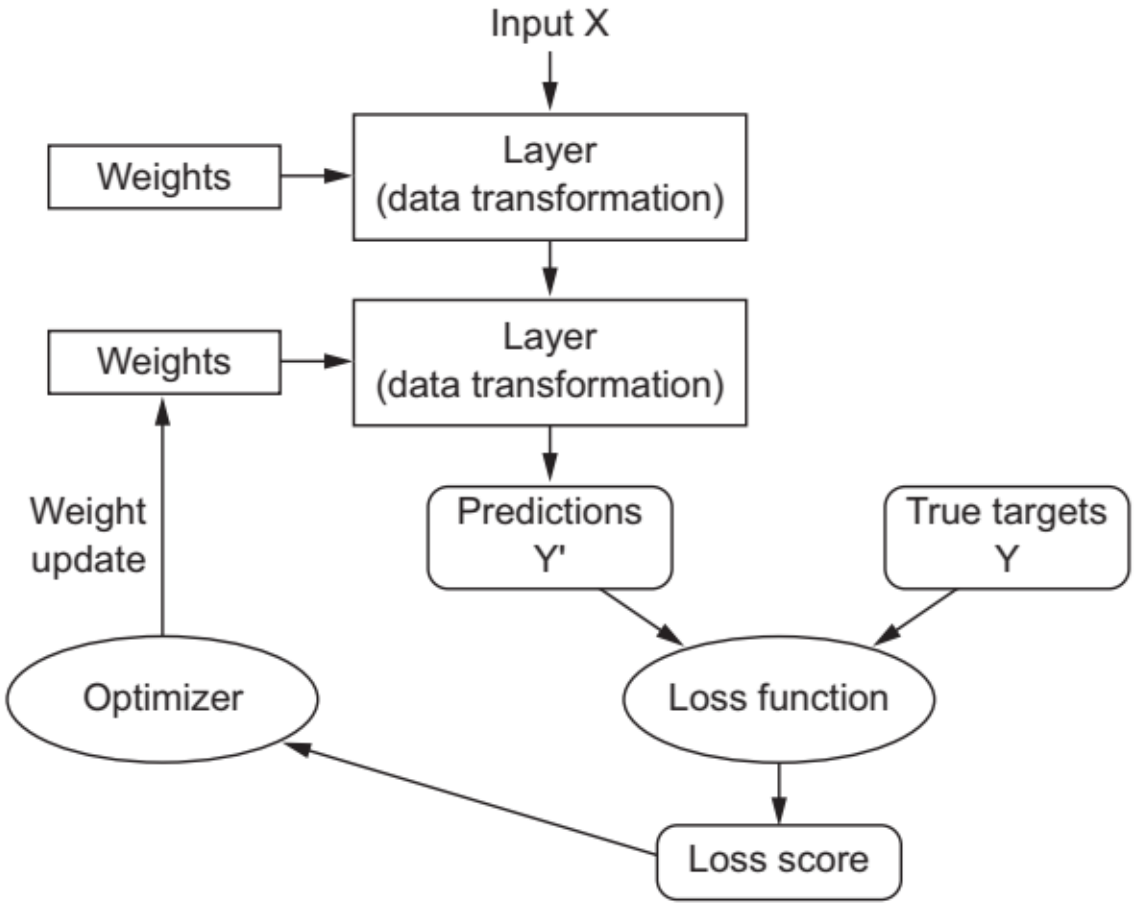


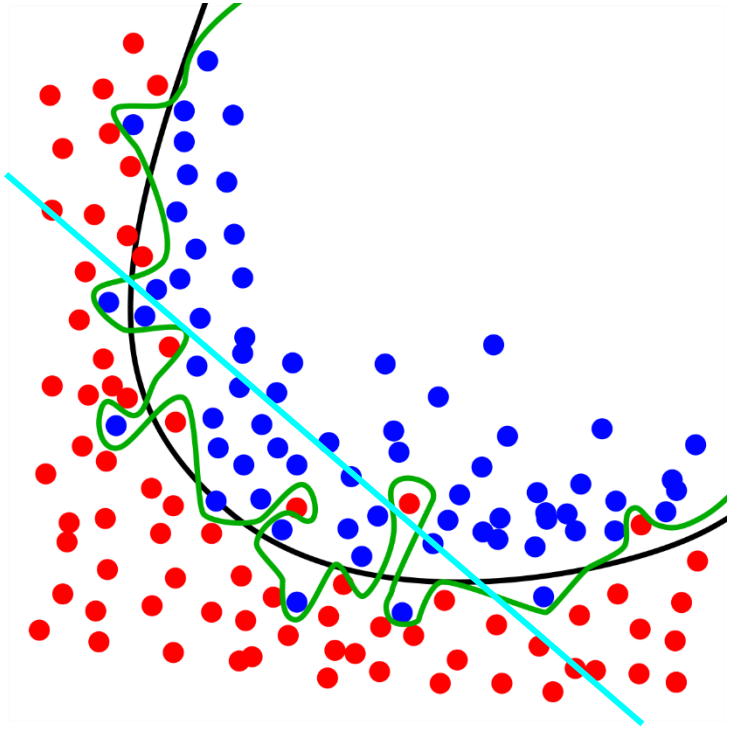
Table 4.1 Choosing the right last-layer activation and loss function for your model

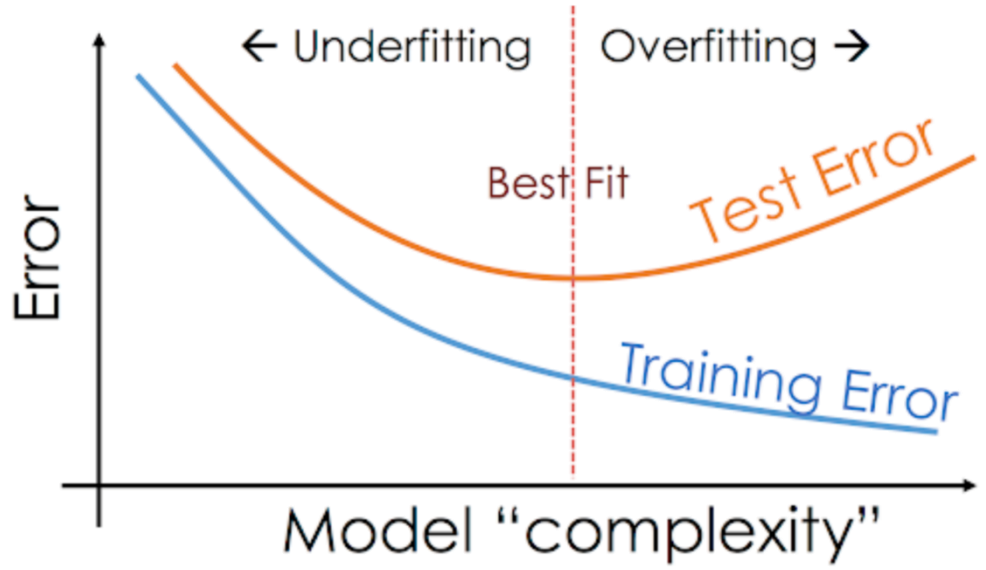
Problem type	Last-layer activation	Loss function
Binary classification	`sigmoid`	`binary_crossentropy`
Multiclass, single-label classification	`softmax`	`categorical_crossentropy`
Multiclass, multilabel classification	`sigmoid`	`binary_crossentropy`
Regression to arbitrary values	None	`mse`
Regression to values between 0 and 1	`sigmoid`	`mse` or `binary_crossentropy`

https://colab.research.google.com/drive/1by92CDbBBTh7Avm_Iy1NNdpPmEN_nF69?usp=sharing

Overfitting and Underfitting

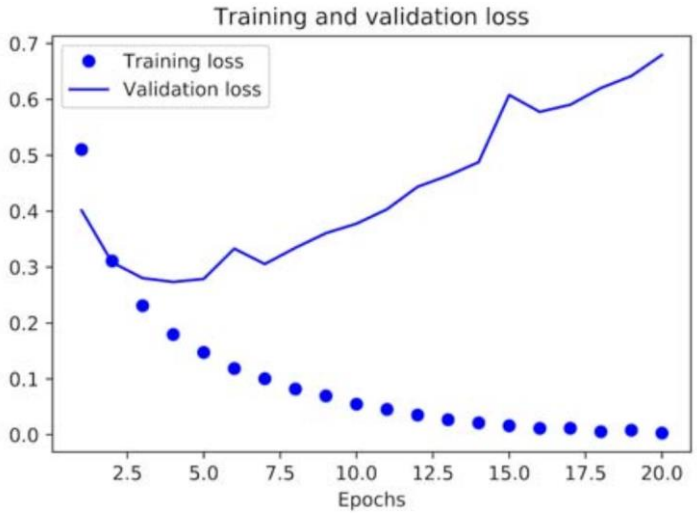
- A central problem in machine learning is how to make an algorithm that will perform well not just on the training data, but also on new inputs

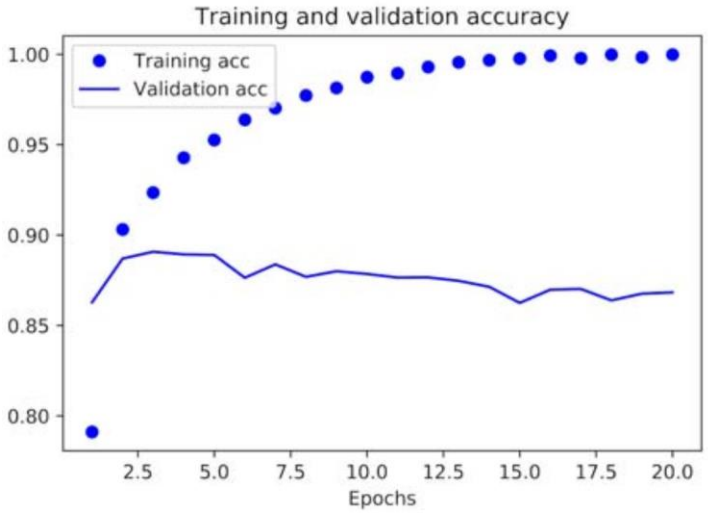


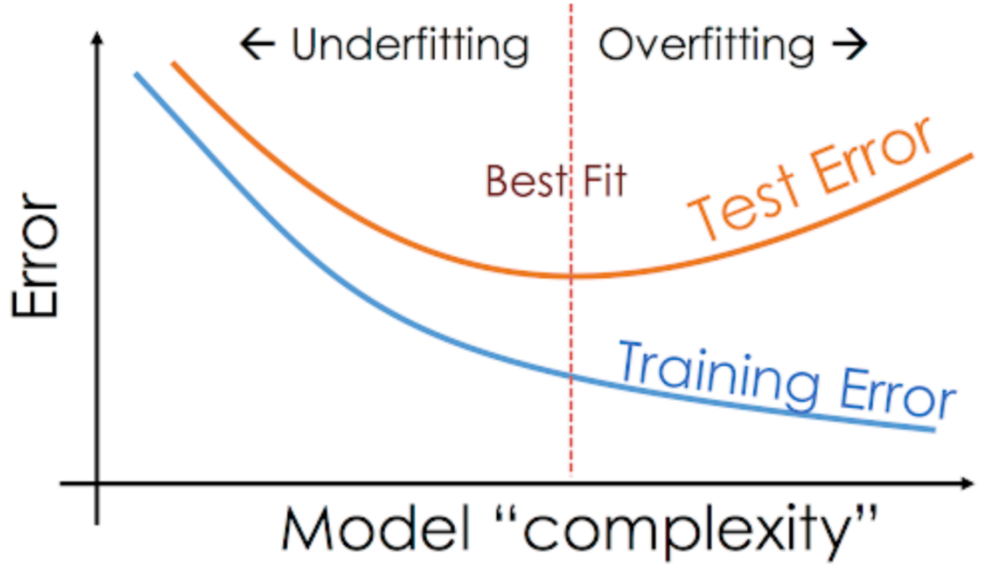


Overfitting and Underfitting

- The performance of the models on the held-out validation data usually peaked after a few epochs and then began to degrade
  - The model quickly started to overfit to the training data
- Learning how to deal with overfitting is essential to mastering ML

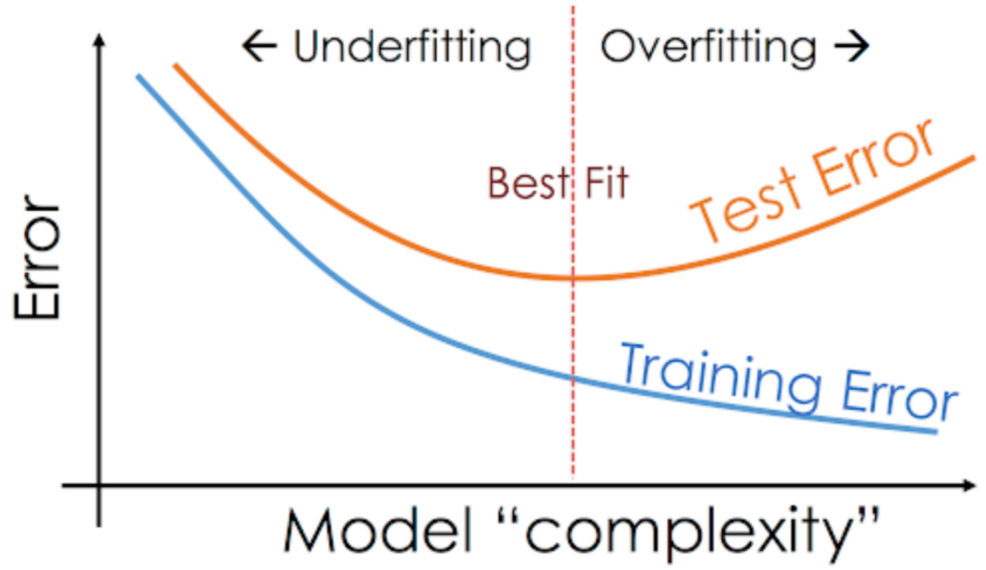






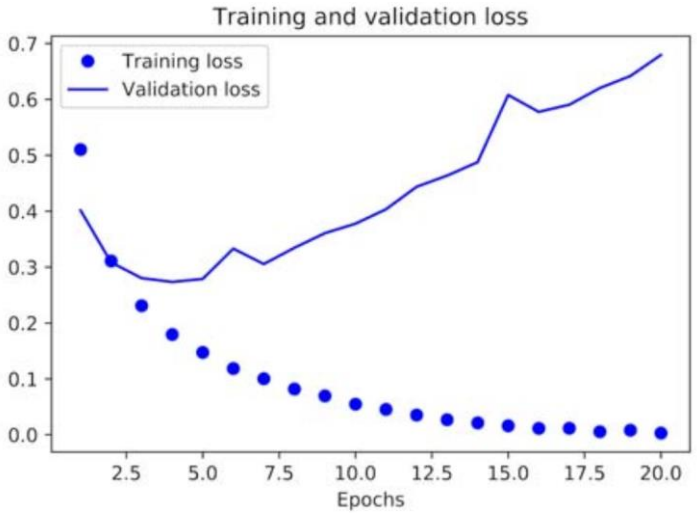
Optimization and Generalization

- Optimization refers to the process of adjusting a model to get the best performance possible on the training data (the learning in ML)
- Generalization refers to how well the trained model performs on data it has never seen before
- The goal is to get good generalization
  - but we don't control generalization!
  - We can only adjust the model based on its training data



Overfitting and Underfitting

- At the beginning of training, optimization and generalization are correlated
  - The model is said to be underfit
  - The network hasn't yet modeled all relevant patterns in the training data
- After some iterations, generalization stops improving, then begin to degrade
  - The model is starting to overfit
  - Learn patterns that are specific to the training data but that are misleading or irrelevant when it comes to new data



Reducing the network's size

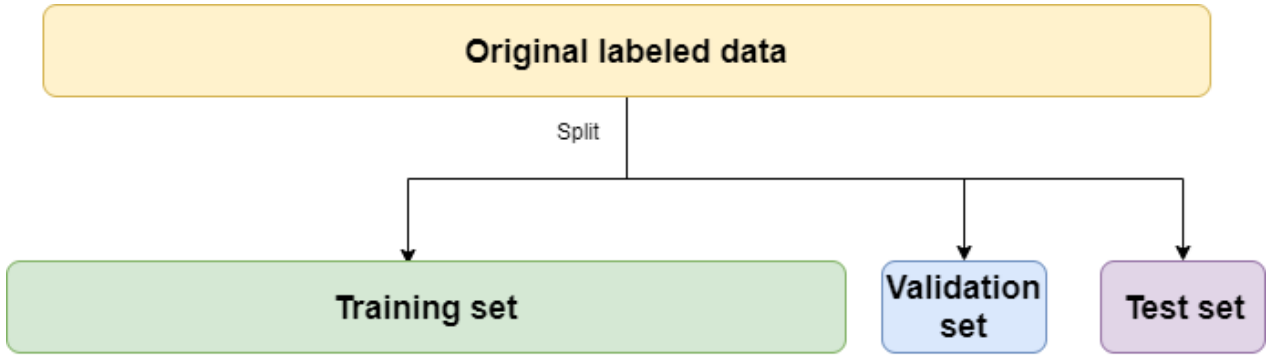
- If a network can only afford to memorize a small number of patterns, the optimization process will force it to focus on the most prominent patterns, which have a better chance of generalizing well
- Intuitively, a model with more parameters has more memorization capacity and therefore can easily learn a perfect dictionary-like mapping between training samples and their targets
- If the network has limited memorization resources, it won't be able to learn this mapping as easily
  - it will have to resort to learning compressed representations that have predictive power
- There is a compromise to be found between too much capacity and not enough capacity

Evaluating machine-learning models

- In machine learning, the goal is to achieve models that generalize—that perform well on never-before-seen data
- It's crucial to be able to reliably measure the generalization power of the model
- Evaluating a model always boils down to splitting the available data into three sets
  - training
  - validation
  - test

Training, validation, and test sets

- Train on the training data (to optimize parameters)
- Evaluate on the validation data (to optimize hyperparameters)
  - 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
  - This tuning is a form of learning
  - Can result in overfitting to the validation set, even though the model is never directly trained on it
- Test it one final time on the test data



Hold-out validation

Listing 4.1 Hold-out validation

```
num_validation_samples = 10000

np.random.shuffle(data)

validation_data = data[:num_validation_samples]
data = data[num_validation_samples:]

training_data = data[:]

model = get_model()
model.train(training_data)
validation_score = model.evaluate(validation_data)

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

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

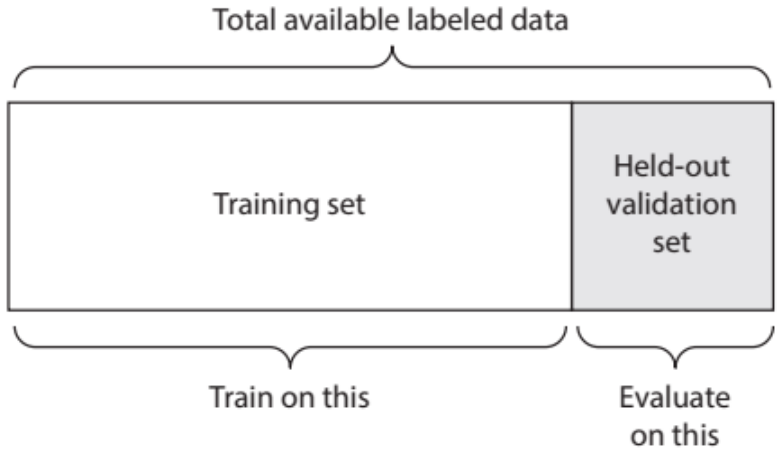
Shuffling the data is usually appropriate.

Defines the validation set

Defines the training set

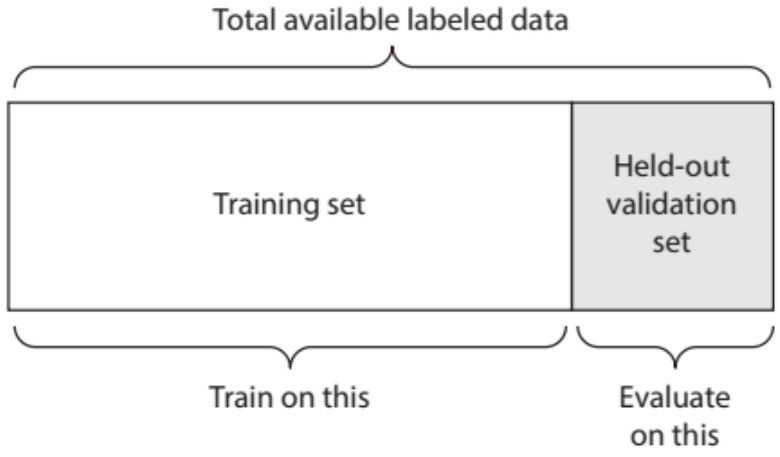
Trains a model on the training data, and evaluates it on the validation data

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



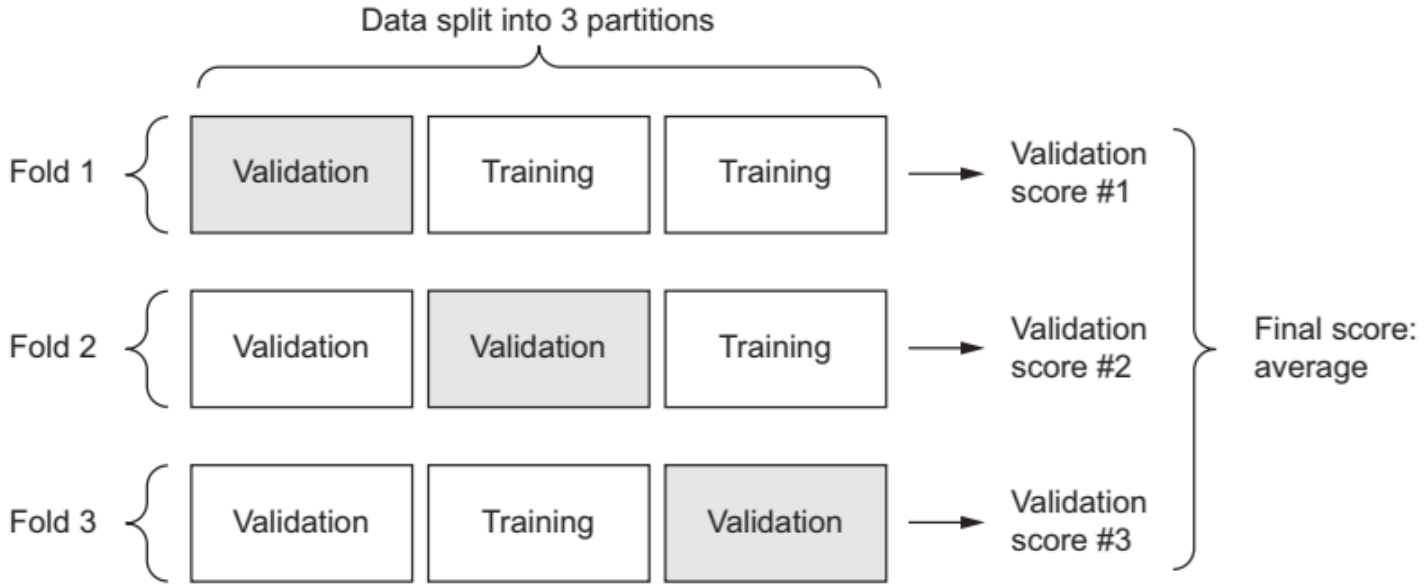
Hold-out validation

- If little data is available, then the validation and test sets may contain too few samples to be statistically representative of the data at hand
- Different random shuffling rounds of the data before splitting end up yielding very different measures of model performance



K-fold cross-validation

- You 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
- Final score is the averages of the K scores



K-fold cross-validation

Listing 4.2 K-fold cross-validation

```
k = 4
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 = data[:num_validation_samples * fold] +
     data[num_validation_samples * (fold + 1):]

    model = get_model()
    model.train(training_data)
    validation_score = model.evaluate(validation_data)
    validation_scores.append(validation_score)

validation_score = np.average(validation_scores)

model = get_model()
model.train(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 is list concatenation, not summation.

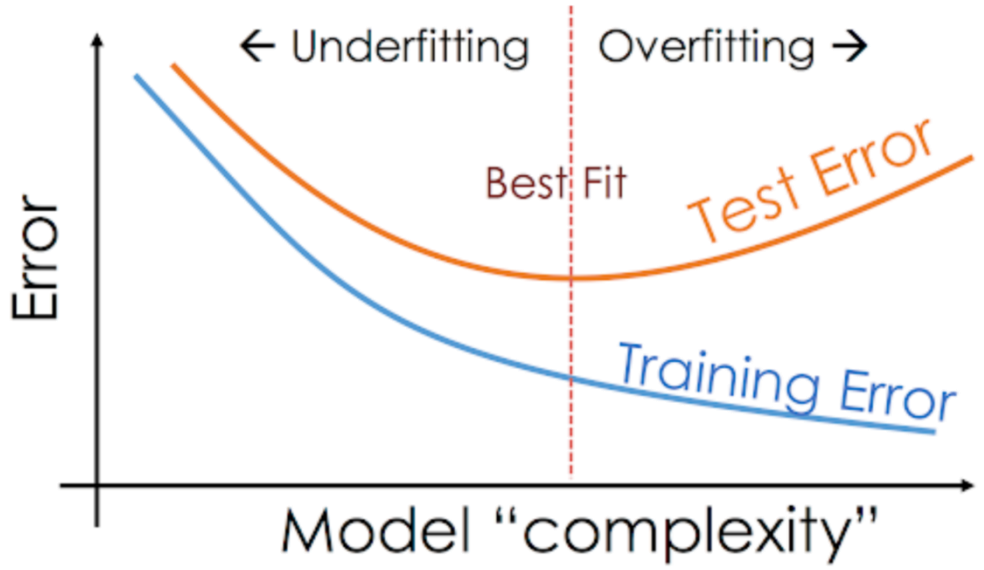
Creates a brand-new instance of the model (untrained)

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

Trains the final model on all non-test data available

Regularization for Deep Learning

- A central problem in machine learning is how to make an algorithm that will perform well not just on the training data, but also on new inputs
- Many strategies used in ML are explicitly designed to reduce the test error, possibly at the expense of increased training error
- These strategies are known as regularization



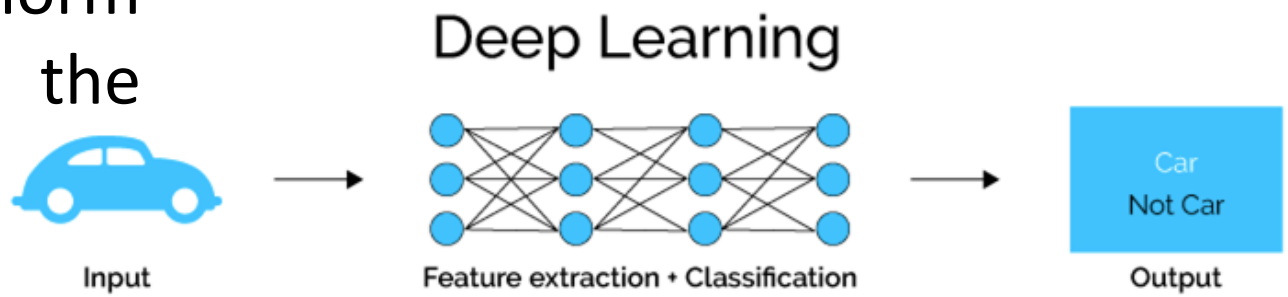
Parameter Norm Penalties

- We can limit the capacity of models by adding a parameter norm penalty to the objective function

$$\tilde{J}(\boldsymbol{\theta}; \boldsymbol{X}, \boldsymbol{y}) = J(\boldsymbol{\theta}; \boldsymbol{X}, \boldsymbol{y}) + \alpha\Omega(\boldsymbol{\theta})$$

- We typically choose a norm penalty Ω that penalizes only the weights of the affine transformation at each layer and leaves the biases unregularized
- We use $\boldsymbol{w}$ to indicate all of the weights that should be affected by a norm penalty, while $\boldsymbol{\theta}$ denotes all of the parameters

$$g\left(\boldsymbol{W}^{(T)}\boldsymbol{x} + \boldsymbol{b}\right)$$



L2 Parameter Regularization

- This regularization strategy drives the weights closer to the origin

$$\Omega(\boldsymbol{w}) = \frac{1}{2}\|\boldsymbol{w}\|_2^2$$

$$\tilde{J}(\boldsymbol{\theta}; \boldsymbol{X}, \boldsymbol{y}) = \frac{\alpha}{2}\boldsymbol{w}^T\boldsymbol{w} + J(\boldsymbol{\theta}; \boldsymbol{X}, \boldsymbol{y})$$

- Gradient of the regularized objective function:

$$\nabla_{\boldsymbol{w}} \tilde{J}(\boldsymbol{\theta}; \boldsymbol{X}, \boldsymbol{y}) = \alpha\boldsymbol{w} + \nabla_{\boldsymbol{w}} J(\boldsymbol{\theta}; \boldsymbol{X}, \boldsymbol{y})$$

$$\boldsymbol{w} \leftarrow \boldsymbol{w} - \epsilon\big(\alpha\boldsymbol{w} + \nabla_{\boldsymbol{w}} J(\boldsymbol{\theta}; \boldsymbol{X}, \boldsymbol{y})\big)$$

$$\boldsymbol{w} \leftarrow (1 - \epsilon\alpha)\boldsymbol{w} - \epsilon\, \nabla_{\boldsymbol{w}} J(\boldsymbol{\theta}; \boldsymbol{X}, \boldsymbol{y})$$