Deep Learning

Week 1: Introduction to Deep Learning

Contents

Introduction

Welcome to the Deep Learning module! Deep learning is one of the most exciting and fastest developing areas of artificial intelligence, and continues to break new ground and set the state of the art in many application domains. In this module you will learn the foundations of deep learning and how to develop neural network architectures for different problem tasks and data domains, including design, initialisation, optimisation, prediction and evaluation.

There will be a particular focus on practical implementation in this module, and for this you will learn the popular Keras Deep Learning API. Keras is a high-level API that is easy to use, whilst still being very flexible and customisable. Since the release of version 3, Keras is a multi-backend framework compatible with TensorFlow, PyTorch or JAX. That means the the Keras API 'wraps' the lower-level operations that are being handled by one of these frameworks. The backend framework can be dynamically selected often without needing to change any code.

These lecture notes are presented in jupyter notebooks to enable integration of Keras code.

This module has been put together using the latest stable release of Keras at the time.

It is likely that the APIs for Keras (and TensorFlow/PyTorch/JAX, and other packages) may change with new version releases, and in the future some of the code presented in these notebooks and the coding tutorials may become deprecated. Make sure to keep an eye on breaking changes in newer library versions.

In this week we will review some important machine learning concepts, including a general definition of machine learning, types of problem tasks, data, and performance measures, generalisation, validation, dataset splits, overfitting/underfitting and methods of regularisation.

We will also get started with Keras by learning about Keras Tensors, which are important low-level objects.

Machine learning recap

Deep learning is a subfield of machine learning, and so many of the core concepts of machine learning will be required in this module. For general machine learning references, see e.g. Bishop, Hastie et al and Murphy. In order to motivate some of the most important concepts, let's first review the definition of machine learning itself. There are several definitions and perspectives on this, but one of the most popular is due to Mitchell:

> A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P if its performance at tasks in T, as measured by P, improves with experience E.

We can unpick this definition by looking at what is meant by *experience E, tasks T* and *performance measure P*.

Tasks T. One of the strengths of deep learning models are their flexibility to solve a wide range of problem tasks. Typical tasks could include:

- Classification
- Regression
- Clustering
- Anomaly detection
- Density estimation

Experience E. This relates to the type of data that is used to accomplish the given task. The data could be labelled examples (such as images of digits and their corresponding labels), unlabelled examples, or streaming data coming from an environment that an agent interacts with (this is the setting for reinforcement learning). Of course, the type of data needs to be appropriate for the learning task. A typical assumption is that the data is independent and identically distributed (iid).

Performance measure P. Given a learning task T and experience E, we then need a way of measuring how well a machine learning system accomplishes the task T. For example, for a regression task this could be the mean squared error, or for a binary classification task we could use binary cross entropy, or area under the ROC curve.

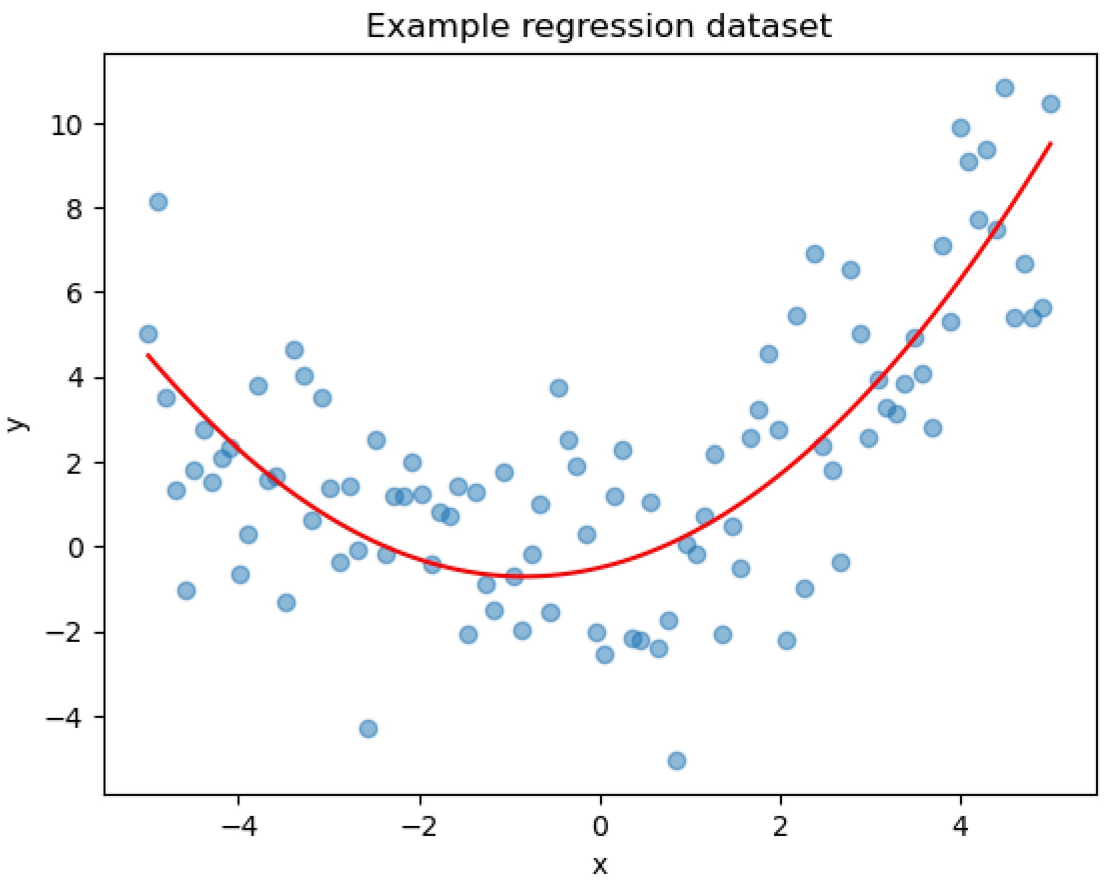
For example, suppose we have a labelled dataset $\mathcal{D} := (x_i, y_i)_{i=1}^N$ of inputs $x_i \in \mathbb{R}^D$ and outputs (or targets) $y_i \in \mathbb{R}$. The task could be a regression task, where we aim to predict a target y given an input x, and where we measure performance using the mean squared error loss. Our aim is to minimise the expected loss under the data distribution:

$$\mathcal{L}(\theta) := \mathbb{E}_{x,y\sim p_{data}}\left[(f_\theta(x) - y)^2\right], \tag{1}$$

where $f_\theta : \mathbb{R}^D \mapsto \mathbb{R}$ is our parameterised regression function, with parameters $\theta \in \mathbb{R}^p$. However, we do not have access to the true data distribution p_{data}, but instead only have access to the set of samples $(x_i, y_i)_{i=1}^N$.

In the following cell we create and plot a toy dataset for illustration.

In [1]:
```
# Create an example toy dataset
# y = 0.3 * x**2 + 0.5 * x - 0.5 + eps,   eps ~ N(0, 4)

import numpy as np
import matplotlib.pyplot as plt

n_samples = 100
x = np.linspace(-5, 5, n_samples)[..., np.newaxis]
y_true = 0.3 * x**2 + 0.5 * x - 0.5
noise = 2 * np.random.randn(n_samples, 1)
y = y_true + noise

plt.plot(x, y_true, 'r-')
plt.scatter(x, y, alpha=0.5)
plt.title("Example regression dataset")
plt.xlabel("x")
plt.ylabel("y")
plt.show()
```



Data splits

In order to obtain a fair measure of the performance of an ML model, we typically split our available data into training and test partitions. The training data is used to infer the optimal parameters of our model, whilst the test data is used purely for evaluation. In addition, a third partition for validation data is also often used to tune the hyperparameters of the model. You should never use the validation or test splits for directly training the model (optimising its parameters).

In the following cell we use `sklearn` to make a training and test partition of our toy dataset.

In [2]:
```python
# We can use the train_test_split from sklearn to conveniently
# split the data

from sklearn.model_selection import train_test_split

print("x shape:", x.shape)
print("y shape:", x.shape)
x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=0.4)
print("\nx_train shape:", x_train.shape)
print("y_train shape:", y_train.shape)
print("\nx_test shape:", x_test.shape)
print("y_test shape:", y_test.shape)
```

```
x shape: (100, 1)
y shape: (100, 1)

x_train shape: (60, 1)
y_train shape: (60, 1)

x_test shape: (40, 1)
y_test shape: (40, 1)
```

This means that in practice what we optimise during training is the loss

$$L(\theta) = \frac{1}{|\mathcal{D}_{train}|} \sum_{x_i, y_i \in \mathcal{D}_{train}} (f_\theta(x_i) - y_i)^2, \qquad (2)$$

where $\mathcal{D}_{train}$ denotes the training data partition.

The following cells illustrate this for our toy dataset, by creating an example regression function and computing the training loss using the inbuilt function from `sklearn`.

In [3]:
```python
# Define an example regression function

def example_f(x_):
    return 0.5 * x_**2 + 0.9 * x_ - 0.25
```

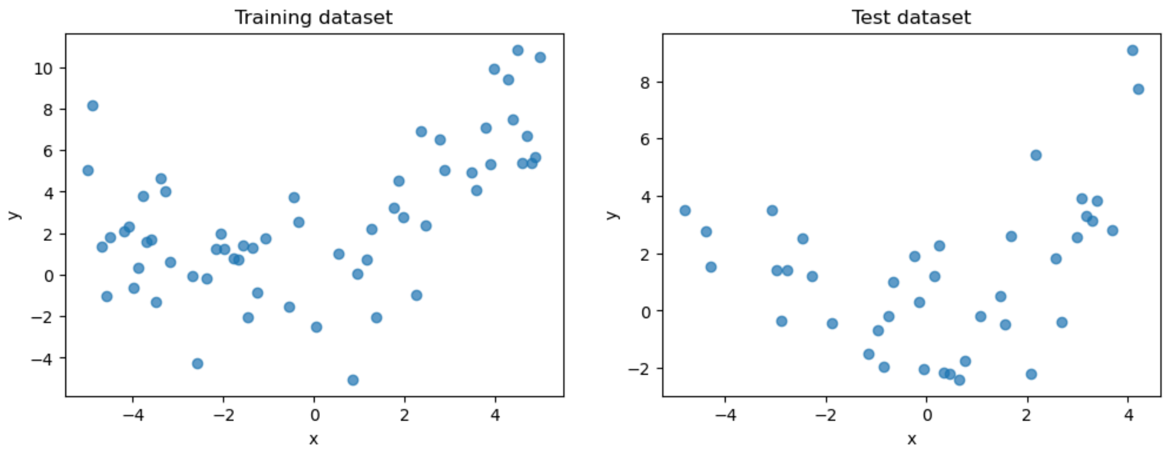
In [ ]:
```python
# Evaluate the MSE on the training set

from sklearn.metrics import mean_squared_error

train_loss = mean_squared_error(example_f(x_train), y_train)
print("Training loss: {:.4f}".format(train_loss))
```

```
Training loss: 15.1875
```

Usually, the bulk of the available data would be used for training. A typical split might be something like 60/20/20 for training/validation/test. In any case, it is important that the validation and test sets are large enough to be a representative sample of the data.

They are both used as proxies for new data that could be presented to the model in deployment, and so give a measure of how well the model generalises to unseen data.

In [5]:
```python
# View the training and test data

fig = plt.figure(figsize=(12, 4))
fig.add_subplot(1, 2, 1)
plt.scatter(x_train, y_train, alpha=0.7)
plt.title("Training dataset")
plt.xlabel("x")
plt.ylabel("y")

fig.add_subplot(1, 2, 2)
plt.scatter(x_test, y_test, alpha=0.7)
plt.title("Test dataset")
plt.xlabel("x")
plt.ylabel("y")
plt.show()
```



Overfitting and underfitting

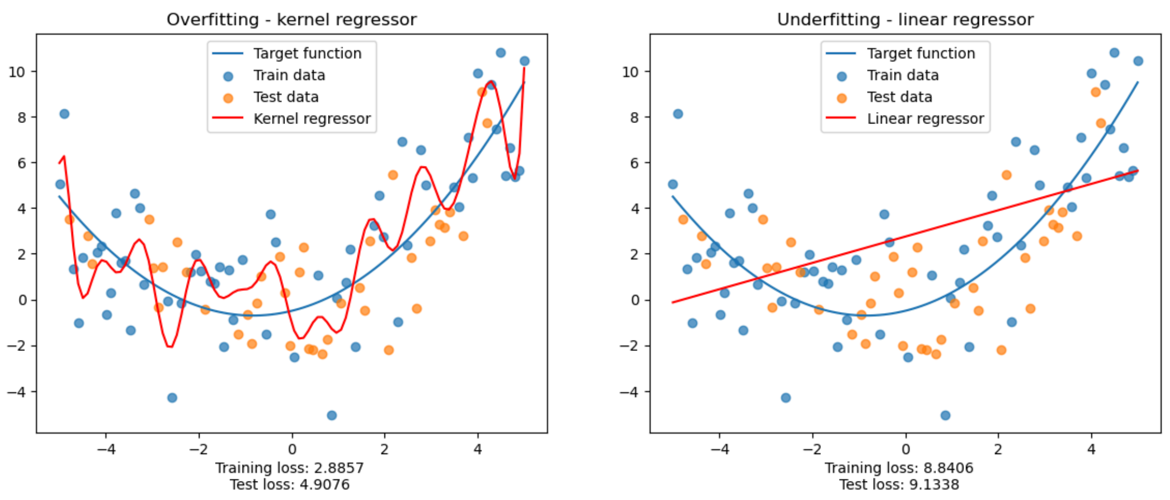
A common issue in all machine learning models is underfitting vs overfitting. Overfitting is when a model becomes too specific to the training data, and doesn't generalise well beyond it. It is characterised by having a good performance measure on the training data and much worse performance on the test data. Underfitting is just the reverse; it is where a model does not manage to fit the training (or test) data well. It is characterised by a similar poor performance on both training and test sets.

The following cells show examples of this using our toy dataset and regression functions from `sklearn`.

In [6]:
```
# Fit a kernel regressor to the training data

from sklearn.kernel_ridge import KernelRidge

kernel_regressor = KernelRidge(alpha=1e-5, kernel='rbf', gamma=1.)
kernel_regressor.fit(x_train, y_train)

kernel_train_loss = mean_squared_error(
    kernel_regressor.predict(x_train),y_train)
kernel_test_loss = mean_squared_error(
    kernel_regressor.predict(x_test), y_test)
```

In [7]:
```
# Fit a high-degree polynomial and a linear model to the training data

from sklearn.linear_model import LinearRegression

linear_regressor = LinearRegression()
linear_regressor.fit(x_train, y_train)

linear_train_loss = mean_squared_error(
    linear_regressor.predict(x_train), y_train)
linear_test_loss = mean_squared_error(
    linear_regressor.predict(x_test), y_test)
```

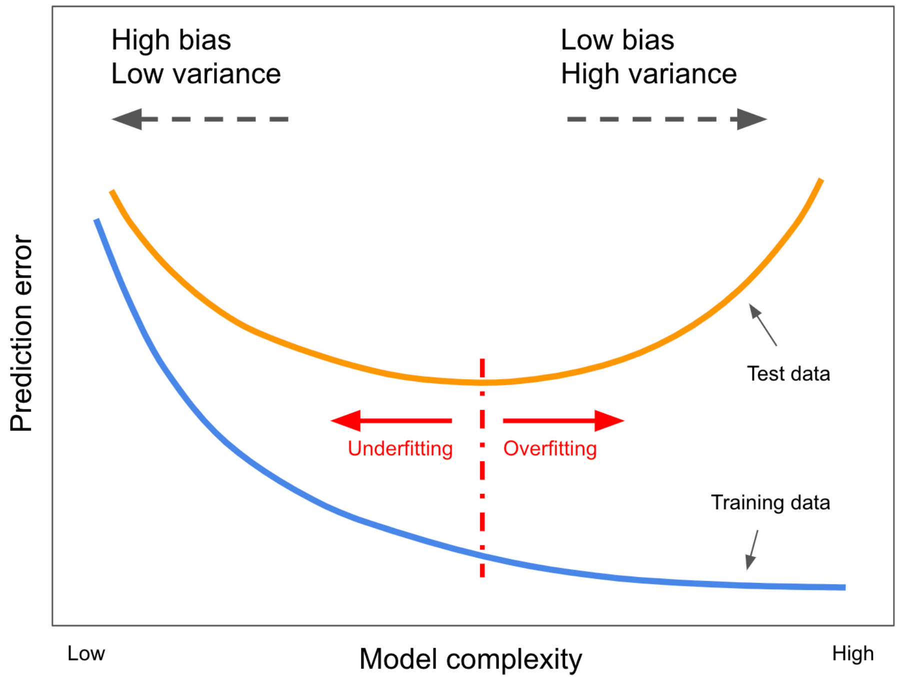
In [8]:
```
# View the training and test data

fig = plt.figure(figsize=(14, 5))
fig.add_subplot(1, 2, 1)
plt.plot(x, y_true, label="Target function")
plt.scatter(x_train, y_train, alpha=0.7, label='Train data')
plt.scatter(x_test, y_test, alpha=0.7, label='Test data')

plt.plot(x, kernel_regressor.predict(x), 'r-', label="Kernel regressor")
plt.title("Overfitting - kernel regressor")
fig.text(0.3, 0.,"Training loss: {:.4f}\nTest loss: {:.4f}".format(
    kernel_train_loss, kernel_test_loss),ha='center')
plt.legend()

fig.add_subplot(1, 2, 2)
plt.plot(x, y_true, label="Target function")
plt.scatter(x_train, y_train, alpha=0.7, label='Train data')
plt.scatter(x_test, y_test, alpha=0.7, label='Test data')

plt.plot(x, linear_regressor.predict(x), 'r-', label="Linear regressor")
plt.title("Underfitting - linear regressor")
fig.text(0.72, 0., "Training loss: {:.4f}\nTest loss: {:.4f}".format(
    linear_train_loss, linear_test_loss),ha='center')
plt.legend()
plt.show()
```



The issue of underfitting vs overfitting is related to **model capacity**, or **model complexity**. These terms refer to the expressive power of a model, or how rich the set of patterns or relationships is that the model class is able to represent. A high capacity model is very expressive, and can represent highly sophisticated relationships in the data. The danger is that if the model capacity is too high, then it can fit spurious relationships or idiosyncrasies that are present in the training data, which aren't representative of the general properties of the data distribution, and this can lead to overfitting. Likewise, if the model capacity is too low, then the model will be unable to represent the patterns or relationships in the data, leading to underfitting.

In our running example, the kernel regressor above has high capacity, whilst the linear regressor has low capacity.

A general schematic plot that we would expect to see is the training loss decreasing as capacity/complexity increases, whilst the test loss will decrease at first but then increase when the model becomes overly complex in relation to the data.

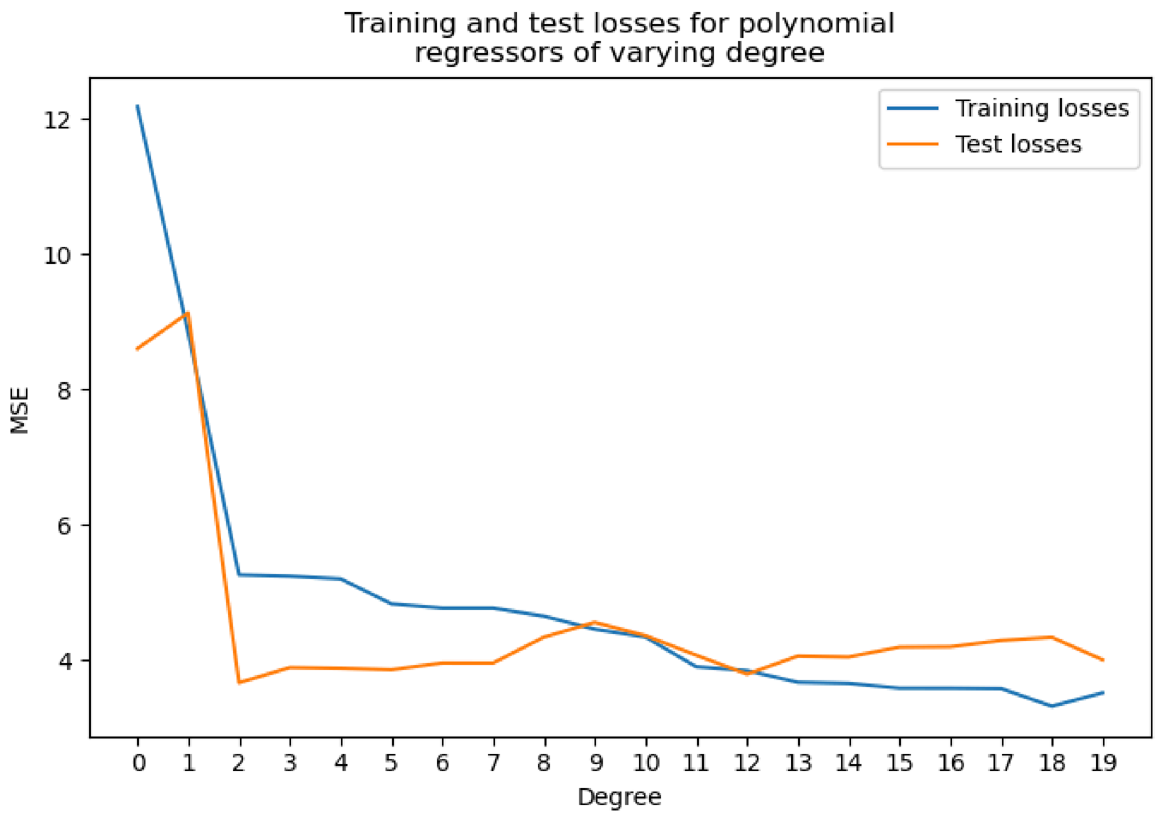


Schematic diagram for error vs complexity; underfitting vs overfitting.

We can see an example of this trend on our running example, if we fit a series of polynomial regressors to the training data with increasing degree. We know that the true underlying function is quadratic, so we would expect to see the test error increasing as the degree increases from 2.

```
In [9]: # Fit polynomial regressors for different degrees

        from sklearn.preprocessing import PolynomialFeatures
        from sklearn.pipeline import make_pipeline
        from sklearn.linear_model import LinearRegression

        num_degrees = 20
        degrees = np.arange(num_degrees)

        polynomial_regressors = []
        for degree in degrees:
            poly_regressor = make_pipeline(PolynomialFeatures(degree),
                                           LinearRegression())
            poly_regressor.fit(x_train, y_train)
            polynomial_regressors.append(poly_regressor)
```

```
In [10]: # Plot the training and test losses for the polynomial regressors

         train_losses = [mean_squared_error(
             pr.predict(x_train), y_train
             ) for pr in polynomial_regressors]
         test_losses = [mean_squared_error(
             pr.predict(x_test), y_test
             ) for pr in polynomial_regressors]

         plt.figure(figsize=(8, 5))
         plt.plot(train_losses, label='Training losses')
         plt.plot(test_losses, label='Test losses')
         plt.title("Training and test losses for polynomial\nregressors of varying
         plt.xticks(np.arange(num_degrees))
         plt.xlabel("Degree")
         plt.ylabel("MSE")
         plt.legend()
         plt.show()
```



Note that if we used the above information to select the best hyperparameter (polynomial degree), then we would in fact be using the data split as a training/validation split, and so should use a separate test set for a final evaluation of the model.

Loss functions

The parameters of deep learning models are often optimised using maximum likelihood estimation. That is, we search for parameters $\hat{\theta}$ such that

$$\hat{\theta} = \arg\max_{\theta} \mathbb{P}_{model}(\mathcal{D}_{train}; \theta).$$

Many common loss functions can be derived through the principle of maximum likelihood and certain modelling assumptions. For example, the mean squared error loss function above (2) can be derived as the negative log-likelihood of the training data under the assumption of homoscedastic Gaussian observation noise (up to a constant scaling factor).

Similarly, cross entropy loss functions arise in classification problems when applying the principle of maximum likelihood. In a supervised classification problem where there are C classes, and we have a parameterised model $f_\theta : \mathbb{R}^D \mapsto [0,1]^C$ whose output can be interpreted as a categorical distribution over C classes, then the categorical cross entropy loss function is given by

$$L(\theta) = -\frac{1}{|\mathcal{D}_{train}|} \sum_{x_i, \tilde{y}_i \in \mathcal{D}_{train}} \sum_{j=1}^{C} \tilde{y}_{ij} \ln f_\theta(x_i)_j,$$

where $\tilde{y}_{ij}$ is the binary label for example i, that is equal to 1 if the correct class is C_j, and 0 otherwise.

In the particular case of binary classification we obtain

$$L(\theta) = -\frac{1}{|\mathcal{D}_{train}|} \sum_{x_i, y_i \in \mathcal{D}_{train}} y_i \ln f_\theta(x_i) + (1 - y_i) \ln(1 - f_\theta(x_i)),$$

where in the above, the labels $y_i \in \{0, 1\}$, $f_\theta : \mathbb{R}^D \mapsto [0, 1]$, and the model output is interpreted as the probability that the input x_i belongs to class 1.

Regularisation

A common way to tackle problems of overfitting is through regularisation methods. Broadly speaking, methods of regularisation seek to constrain the model capacity. There are many methods of regularisation depending on the model, data and task. Some examples of regularisation techniques that are used for deep learning models are:

- Weight decay
- Patience/early stopping
- Control model complexity
- Dropout
- Weight sharing
- Dataset augmentation

We will be using several of these regularisation techniques in implementations during the module. The last of these - dataset augmentation - is worth singling out as the best form of regularisation if it is possible.

For example, for a linear model of the form

$$f(\mathbf{x}) = \sum_j w_j \phi_j(\mathbf{x})$$

a typical regularisation is to add a sum of squares penalty term to discourage the weights w_j from getting too large. Specifically, instead of minimising the mean squared error (2), we minimise the loss function

$$L(\mathbf{w}, \alpha) = \frac{1}{|\mathcal{D}_{train}|} \left(\sum_{x_i, y_i \in \mathcal{D}_{train}} (f(x_i) - y_i)^2 + \alpha \sum_j w_j^2 \right), \qquad (3)$$

where the coefficient α is a hyperparameter.

Tensors and operations

Tensors are core objects inside a Keras model. You can think of Tensors as being multidimensional versions of vectors and arrays. When we build our neural network models, what we're doing is defining a computational graph, where input data is processed through the layers of the network and sent through the graph all the way to the outputs. Tensors are the objects that get passed around within the graph, and capture those computations within the graph.

Keras Tensors and operations wrap corresponding lower-level objects that are handled in the backend framework. In this section we will introduce these Tensors, and some example operations on them.

First, let's configure the Keras backend to select the underlying framework. This must be done before importing Keras. One way is to set the environment variable `KERAS_BACKEND`, which can be done within the python program as follows:

```
In [11]: # Configure the backend

import os

os.environ["KERAS_BACKEND"] = "torch"
# Options are "tensorflow", "torch" or "jax"
```

Another option is to edit the config file stored at `~/.keras/keras.json`. See this guide for more information.

Most of the time it won't matter which backend you select, as the Keras code will remain the same (try re-running the code in this section with different backends). Occasionally, we will need to go a little lower level and write backend-specific code (for example, when customising training algorithms). For this purpose, we will cover the basics of both TensorFlow and PyTorch, and you are free to use either of these backends throughout the course.

Now let's import Keras and check the backend we're using. The Keras version can also be checked with `keras.version()`.

```
In [12]: # Check the backend

import keras

keras.config.backend()
```

```
Out[12]: 'torch'
```

A Tensor object can be created from a list or numpy array as follows.

```
In [13]: # Create a constant Tensor

a = keras.ops.array([1, 2, 3])
print(a)
```

```
tensor([1, 2, 3], device='cuda:0', dtype=torch.int32)
```

Note that the object that is returned above will depend on the selected backend.

We can see that Tensors have `shape` and `dtype` properties, similar to NumPy arrays.

In [14]:
```
# Examine shape property

a.shape
```

Out[14]:
```
torch.Size([3])
```

In [15]:
```
# Examine dtype property

a.dtype
```

Out[15]:
```
torch.int32
```

Tensor objects can have different types, just like NumPy arrays.

In [16]:
```
# Create Tensor objects of different type

import numpy as np

bool_array = np.array([True, False], dtype=bool)
float_array = np.array([3.14159, 2.71828], dtype=np.float32)

bool_tensor = keras.ops.convert_to_tensor(bool_array)
float_tensor  = keras.ops.convert_to_tensor(float_array)
print(bool_tensor, bool_tensor.dtype)
print(float_tensor, float_tensor.dtype)
```

```
tensor([ True, False], device='cuda:0') torch.bool
tensor([3.1416, 2.7183], device='cuda:0') torch.float32
```

We can convert a Tensor into a NumPy array using the `convert_to_numpy` function.

In [17]:
```
# Convert Tensor to NumPy array

a_np = keras.ops.convert_to_numpy(a)
print(type(a_np))
a_np
```

```
<class 'numpy.ndarray'>
```

Out[17]:
```
array([1, 2, 3], dtype=int32)
```

In [18]:
```
# Create a rank-2 Tensor

b = keras.ops.array([[1.2, 0.4, 0.7], [-9.3, 4.5, 1.1]])
b
```

Out[18]:
```
tensor([[ 1.2000,  0.4000,  0.7000],
        [-9.3000,  4.5000,  1.1000]], device='cuda:0')
```

In [19]:
```
# Get Tensor rank

print(a.ndim)
print(b.ndim)
```

```
1
2
```

In [20]:
```
# Create a Tensor with tf.ones
```

```
keras.ops.ones((2, 2))
```

```
Out[20]:  tensor([[1., 1.],
                  [1., 1.]], device='cuda:0')
```

```
In [21]: # Create a Tensor with tf.zeros

         keras.ops.zeros((3,))
```

```
Out[21]:  tensor([0., 0., 0.], device='cuda:0')
```

```
In [22]: # Change the type of a Tensor

         keras.ops.cast(keras.ops.zeros((3,)), 'int')
```

```
Out[22]:  tensor([0, 0, 0], device='cuda:0', dtype=torch.int32)
```

We can compute Tensor multiplication using `keras.ops.tensordot` (see the docs). The `axes` argument can be an integer or list of integers. When it is a single integer `n`, the contraction is performed over the last `n` axes of the first Tensor and the first `n` axes of the second Tensor. If it is a list, then the elements of the list specify the axes to contract.

```
In [23]: # Compute matrix-matrix product

         c = keras.ops.array([[1.2, 3.4],
                              [5.6, 7.8]])
         d = keras.ops.array([[-1.0, -0.5],
                              [0.5, 1.0]])

         print(keras.ops.tensordot(c, d, axes=1))
         # Sum over last axis of c and first axis of d
         print(keras.ops.tensordot(c, d, axes=[[1], [0]]))  # Equivalent
```

```
tensor([[ 0.5000,  2.8000],
        [-1.7000,  5.0000]], device='cuda:0')
tensor([[ 0.5000,  2.8000],
        [-1.7000,  5.0000]], device='cuda:0')
```

```
In [24]: # Compute matrix-vector product

         print(keras.ops.tensordot(b, a, axes=1))
         print(keras.ops.tensordot(b, a, axes=[[1], [0]]))
```

```
tensor([4.1000, 3.0000], device='cuda:0')
tensor([4.1000, 3.0000], device='cuda:0')
```

For both rank-1 and rank-2 Tensors, we can use the `keras.ops.matmul` function. `keras.ops.matmul` will return the dot product if the two inputs are 1-dimensional, and will return the matrix multiplication of the two inputs otherwise. (For details, see the docs.)

```
In [25]: # Use keras.ops.matmul to compute product

         print(keras.ops.matmul(b, a))
```

```
tensor([4.1000, 3.0000], device='cuda:0')
```

Useful operations to manipulate Tensor shapes are `keras.ops.expand_dims`, `keras.ops.squeeze` and `keras.ops.reshape`.

In [26]:
```
# Add an extra dimension to a Tensor

a = keras.ops.expand_dims(a, 1)
print(keras.ops.shape(a))
```

```
(3, 1)
```

In [27]:
```
# Use keras.ops.matmul, keras.ops.squeeze and keras.ops.reshape

keras.ops.reshape(keras.ops.squeeze(keras.ops.matmul(b, a)), [1, 2])
```

Out[27]:
```
tensor([[4.1000, 3.0000]], device='cuda:0')
```

Random Tensors

Keras also has the RNG API for random number operations. For example, it is also often useful to fill Tensors with random values.

In [28]:
```
# Create a random normal Tensor

keras.random.normal((3, 3))
```

Out[28]:
```
tensor([[ 0.5606,  0.2395, -0.4507],
        [-0.5241,  1.0656, -0.8956],
        [-0.7713, -0.5362, -0.1462]], device='cuda:0')
```

In [29]:
```
# Create a random integer Tensor

keras.random.randint(shape=(2, 4), minval=0, maxval=10)
```

Out[29]:
```
tensor([[5, 0, 1, 5],
        [0, 6, 5, 4]], device='cuda:0', dtype=torch.int32)
```

In [30]:
```
# Create a random uniform Tensor

keras.random.uniform(shape=(2, 4), minval=0, maxval=10)
```

Out[30]:
```
tensor([[9.8146, 1.8123, 0.9553, 7.1973],
        [0.2636, 4.5007, 7.8306, 4.1682]], device='cuda:0')
```

We will be using these operations and more to manipulate Tensors as we develop our deep learning models.

Keras Variables

Keras Tensors are normally used to store the result of operations carried out in the computational graph that represents your deep learning model. Keras Variables are used to store the values of *stateful* Tensors, such as parameters of the model that may change over the course of training.

In [31]:
```
# Create a Keras Variable

initial_value = keras.random.normal((2, 2))
```

```
u = keras.Variable(initial_value)
u
```

```
Out[31]: <Variable path=variable, shape=(2, 2), dtype=float32, value=[[-0.1663545
         4 -0.6885658 ]
          [-0.04060998 -0.6043903 ]]>
```

To see the values that the Variable holds, use the `.value` attribute.

```
In [32]: # View the Variable value

         u.value
```

```
Out[32]: Parameter containing:
         tensor([[-0.1664, -0.6886],
                 [-0.0406, -0.6044]], device='cuda:0', requires_grad=True)
```

Unlike Keras Tensors, Keras Variables come with extra methods for updating their state. These are `assign`, `assign_add` and `assign_sub`.

```
In [33]: # Assign a new value to the Variable

         new_value = 2. * keras.ops.ones((2, 2))
         u.assign(new_value)
         u.value
```

```
Out[33]: Parameter containing:
         tensor([[2., 2.],
                 [2., 2.]], device='cuda:0', requires_grad=True)
```

```
In [34]: # Add a value to the Variable

         increment = keras.ops.array([[0., 0.], [1., 1.]])
         u.assign_add(increment)
         u.value
```

```
Out[34]: Parameter containing:
         tensor([[2., 2.],
                 [3., 3.]], device='cuda:0', requires_grad=True)
```

```
In [35]: # Subtract a value from the Variable

         decrement = keras.ops.array([[2., 0.], [2., 0.]])
         u.assign_sub(decrement)
         u.value
```

```
Out[35]: Parameter containing:
         tensor([[0., 2.],
                 [1., 3.]], device='cuda:0', requires_grad=True)
```

We will often use Variables in operations within the computational graph. The result of the operation is a Tensor.

```
In [36]: # Use a Variable in a simple operation

         v = keras.Variable([2.6, -0.4])
         s = v + 1
         s
```

```
Out[36]:  tensor([3.6000, 0.6000], device='cuda:0', grad_fn=<AddBackward0>)
```

Check the docs!

The Keras documentation should be a resource that you regularly use throughout this module. You will find further details on functions and classes that are covered, and will likely need to refer back to this resource to check correct usage, as well as to find additional functions and methods to suit your needs.

In this section we have looked at parts of the Ops API, which is used to operate and manipulate Tensors. It has the namespace `keras.ops`. It includes Core Ops, RNG API and NumPy Ops, which provides a range of functions that will be familiar to NumPy users.

You can also find many useful developer guides and code examples to gain a deeper understanding of Keras and explore additional use cases.

References

- Bishop, C. M. (2006), "Pattern Recognition and Machine Learning", Springer-Verlag, Berlin, Heidelberg.
- Clevert, D.-A., Unterthiner, T., & Hochreiter, S. (2016), "Fast and Accurate Deep Network Learning by Exponential Linear Units (ELUs)", in *4th International Conference on Learning Representations, {ICLR} 2016", San Juan, Puerto Rico, May 2-4, 2016.
- Hastie, T., Tibshirani, R. & Friedman, J. (2001), "The Elements of Statistical Learning", Springer New York Inc., New York, NY, USA.
- Klambauer, G., Unterthiner, T., Mayr, A., & Hochreiter, S. (2017), "Self-Normalizing Neural Networks", *Neural Information Processing Systems (NIPS)*, 971-980.
- McCulloch, W. & Pitts, W. (1943), "A Logical Calculus of Ideas Immanent in Nervous Activity", Bulletin of Mathematical Biophysics, **5**, 127-147.
- Mitchell, T. (1997), "Machine Learning", McGraw-Hill, New York.
- Murphy, K. P. (2012), "Machine Learning: A Probabilistic Perspective", The MIT Press.
- Ramachandran, P., Zoph, B. & Le, Q. V. (2018) "Searching for Activation Functions", arXiv preprint, abs/1710.05941.
- Rosenblatt, F. (1958), "The Perceptron: A Probabilistic Model for Information Storage and Organization in The Brain", Psychological Review, 65-386.
- Rosenblatt, F. (1961), "Principles of Neurodynamics. Perceptrons and the Theory of Brain Mechanisms", Defense Technical Information Center.