Module 1

Introduction to ANNs with Keras

Introduction

- Machine learning models are inspired by the networks of biological neurons found in our brains.
- ANNs have gradually become quite different from their biological cousins.
- Some researchers even argue that we should drop the biological analogy altogether (e.g., by saying “units” rather than “neurons”)

Introduction

- ANNs are versatile, powerful, and scalable, making them ideal to tackle large and highly complex machine learning tasks such as
  - Classifying billions of images (e.g., Google Images)
  - Powering speech recognition services (e.g., Apple’s Siri)
  - Recommending the best videos to watch to hundreds of millions of users every day (e.g., YouTube), or
  - Learning to beat the world champion at the game of Go (DeepMind’s AlphaGo).

From Biological to Artificial Neurons

- ANNs were first introduced back in 1943 by the neurophysiologist Warren McCulloch and the mathematician Walter Pitts.
- McCulloch and Pitts presented a simplified computational model of how biological neurons might work together in animal brains to perform complex computations.
- The early successes of ANNs led to the widespread belief that we would soon be conversing with truly intelligent machines. When it became clear in the 1960s that this promise would go unfulfilled, funding flew elsewhere, and ANNs entered a long winter.

From Biological to Artificial Neurons

- In the early 1980s, new architectures were invented and better training techniques were developed, sparking a revival of interest in the study of neural networks. But progress was slow, and by the 1990s other powerful machine learning techniques had been invented, such as support vector machines. These techniques seemed to offer better results and stronger theoretical foundations than ANNs, so once again the study of neural networks was put on hold.
- We are now witnessing yet another wave of interest in ANNs.

From Biological to Artificial Neurons

Here are a few good reasons to believe that this time is different and that the renewed interest in ANNs will have a much more profound impact on our lives:

- There is now a huge quantity of data available to train neural networks, and ANNs frequently outperform other ML techniques on very large and complex problems.
- The tremendous increase in computing power since the 1990s now makes it possible to train large neural networks in a reasonable amount of time.
- The training algorithms have been improved.

The Perceptron

The *perceptron* is one of the simplest ANN architectures, invented in 1957 by Frank Rosenblatt.

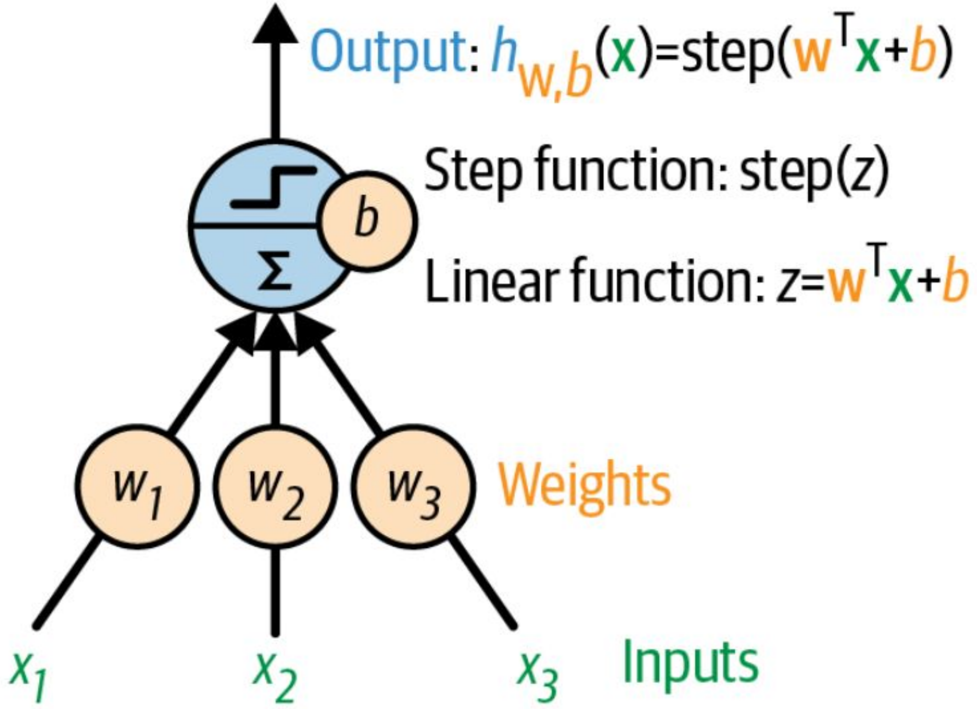
It is based on a slightly different artificial neuron called a Threshold Logic Unit (TLU)

The inputs are numbers (instead of binary on/off values), and each input connection is associated with a weight.

The TLU first computes a linear function of its inputs and then it applies a *step function* to the result.

The Perceptron

The TLU first computes a linear function of its inputs and then it applies a *step function* to the result.


Output: $h_{\mathbf{w},b}(\mathbf{x})=\text{step}(\mathbf{w}^T\mathbf{x}+b)$
Step function: step(z)
b
Σ
Linear function: $z=\mathbf{w}^T\mathbf{x}+b$
w_1
w_2
w_3
Weights
x_1
x_2
x_3
Inputs

The Perceptron

The most common step function used in perceptrons is the *Heaviside step function*. Sometimes the sign function is used instead.

$$\text{heaviside}(z) = \begin{cases} 0 & \text{if } z < 0 \\ 1 & \text{if } z \geq 0 \end{cases}$$

$$\text{sgn}(z) = \begin{cases} -1 & \text{if } z < 0 \\ 0 & \text{if } z = 0 \\ +1 & \text{if } z > 0 \end{cases}$$

The Perceptron

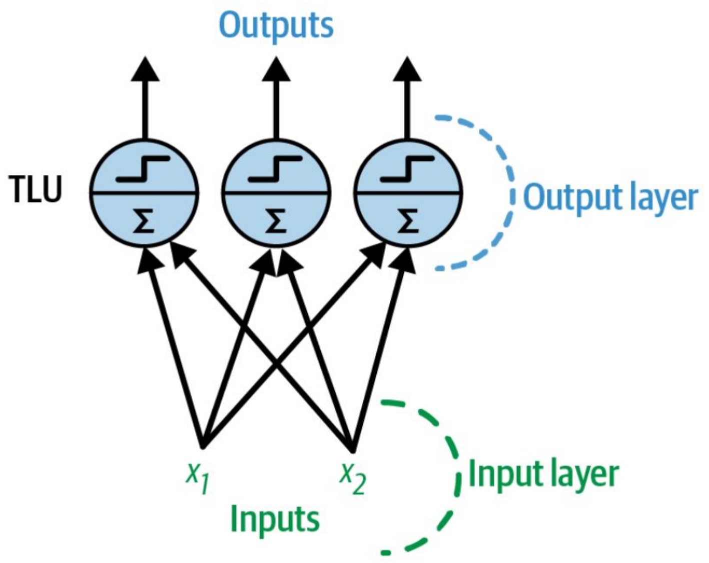
A perceptron is composed of one or more TLUs organized in a single layer, where every TLU is connected to every input.

Such a layer is called a *fully connected layer*, or a *dense layer*.

The inputs constitute the *input layer*.

And since the layer of TLUs produces the final outputs, it is called the *output layer*.

A perceptron with 2 inputs and 3 outputs is shown



The Perceptron

How is a perceptron trained?

The perceptron training algorithm proposed by Rosenblatt was largely inspired by *Hebb's rule*. Donald Hebb suggested that when a biological neuron triggers another neuron often, the connection between these two neurons grows stronger. "Cells that fire together, wire together"

Perceptrons are trained using a variant of this rule that takes into account the error made by the network when it makes a prediction; the perceptron learning rule reinforces connections that help reduce the error. For every output neuron that produced a wrong prediction, it reinforces the connection weights from the inputs that would have contributed to the correct prediction.

The Perceptron

The decision boundary of each output neuron is linear, so perceptrons are incapable of learning complex patterns (just like logistic regression classifiers).

However, if the training instances are linearly separable, Rosenblatt demonstrated that this algorithm would converge to a solution.

This is called the ***perceptron convergence theorem***.

The Perceptron

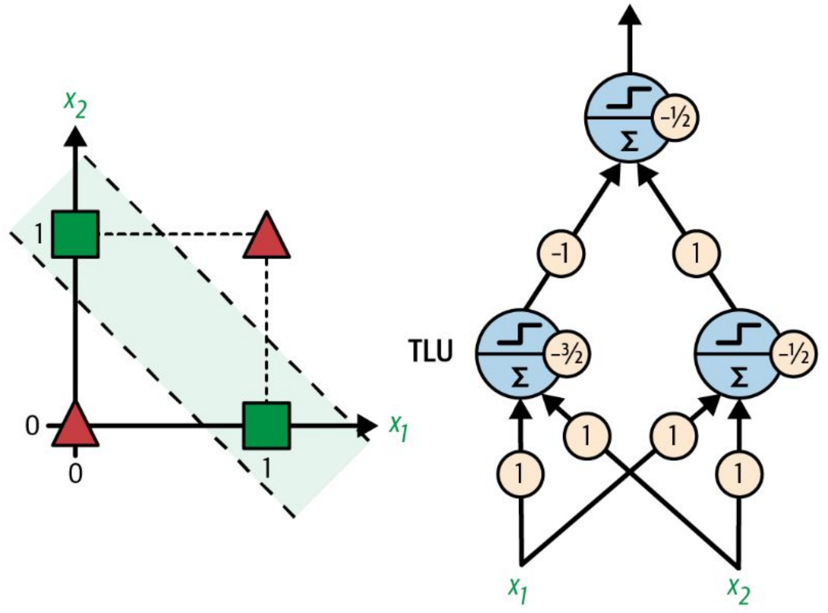
In 1969, Marvin Minsky and Seymour Papert highlighted a number of serious weaknesses of perceptrons - in particular, the fact that they are incapable of solving some trivial problems (e.g., the *exclusive OR* (XOR) classification problem).

This is true of any other linear classification model (such as logistic regression classifiers), but researchers had expected much more from perceptrons, and some were so disappointed that they dropped neural networks altogether.

The Perceptron

It turns out that some of the limitations of perceptrons can be eliminated by stacking multiple perceptrons => *multilayer perceptron* (MLP).

An MLP can solve the XOR problem: with inputs (0, 0) or (1, 1), the network outputs 0, and with inputs (0, 1) or (1, 0) it outputs 1.

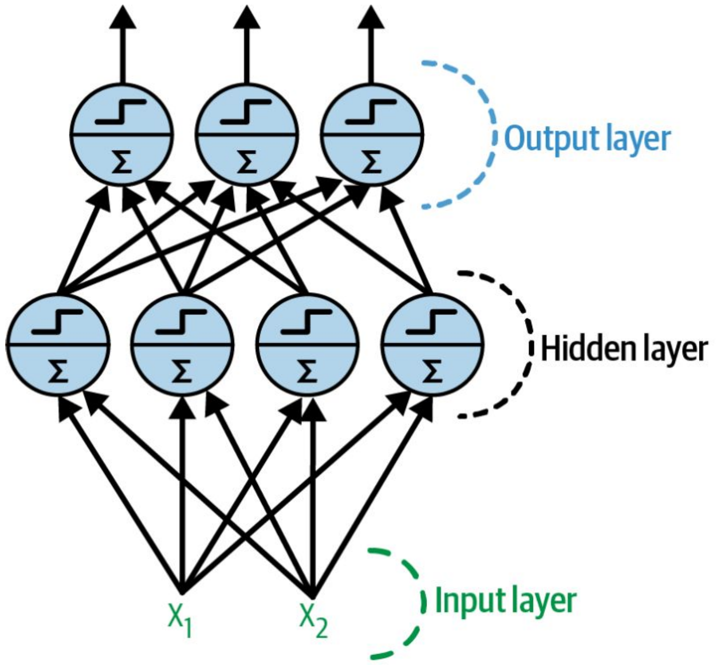


The Perceptron

Contrary to logistic regression classifiers, perceptrons do not output a class probability. This is one reason to prefer logistic regression over perceptrons.

The Multilayer Perceptron and Backpropagation

An MLP is composed of one input layer, one or more layers of TLUs called *hidden layers*, and one final layer of TLUs called the *output layer*



The signal flows only in one direction (from the inputs to the outputs), so this architecture is an example of a *feedforward neural network* (FNN).

Deep Neural Network

When an ANN contains a deep stack of hidden layers, it is called a *deep neural network* (DNN).

For many years researchers struggled to find a way to train MLPs, without success. In the early 1960s several researchers discussed the possibility of using gradient descent to train neural networks, but this requires computing the gradients of the model’s error with regard to the model parameters; it wasn’t clear at the time how to do this efficiently with such a complex model containing so many parameters, especially with the computers they had back then.

Backpropagation

In 1970, a researcher named Seppo Linnainmaa introduced in his master’s thesis a technique to compute all the gradients automatically and efficiently. This algorithm is now called *reverse-mode automatic differentiation* (or *reverse-mode auto- diff* for short). In just two passes through the network (one forward, one backward), it is able to compute the gradients of the neural network’s error with regard to every single model parameter.

In other words, it can find out how each connection weight and each bias should be tweaked in order to reduce the neural network’s error. These gradients can then be used to perform a gradient descent step. If you repeat this process of computing the gradients automatically and taking a gradient descent step, the neural network’s error will gradually drop until it eventually reaches a minimum. This combination of reverse-mode autodiff and gradient descent is now called *backpropagation* (or *backprop* for short).

Backpropagation

1. It handles one mini-batch at a time (for example, containing 32 instances each), and it goes through the full training set multiple times. Each pass is called an *epoch*.
2. Each mini-batch enters the network through the input layer. The algorithm then computes the output of all the neurons in the first hidden layer, for every instance in the mini-batch. The result is passed on to the next layer, its output is computed and passed to the next layer, and so on until we get the output of the last layer, the output layer. This is the *forward pass*: it is exactly like making predictions, except all intermediate results are preserved since they are needed for the backward pass.
3. Next, the algorithm measures the network’s output error (i.e., it uses a loss function that compares the desired output and the actual output of the network, and returns some measure of the error).

Backpropagation

4. Then it computes how much each output bias and each connection to the output layer contributed to the error. This is done analytically by applying the *chain rule* (perhaps the most fundamental rule in calculus), which makes this step fast and precise.

5. The algorithm then measures how much of these error contributions came from each connection in the layer below, again using the chain rule, working backward until it reaches the input layer. As explained earlier, this reverse pass efficiently measures the error gradient across all the connection weights and biases in the network by propagating the error gradient backward through the network (hence the name of the algorithm).

6. Finally, the algorithm performs a gradient descent step to tweak all the connection weights in the network, using the error gradients it just computed.

Backpropagation

In short, backpropagation makes predictions for a mini-batch (forward pass), measures the error, then goes through each layer in reverse to measure the error contribution from each parameter (reverse pass), and finally tweaks the connection weights and biases to reduce the error (gradient descent step).

Backpropagation

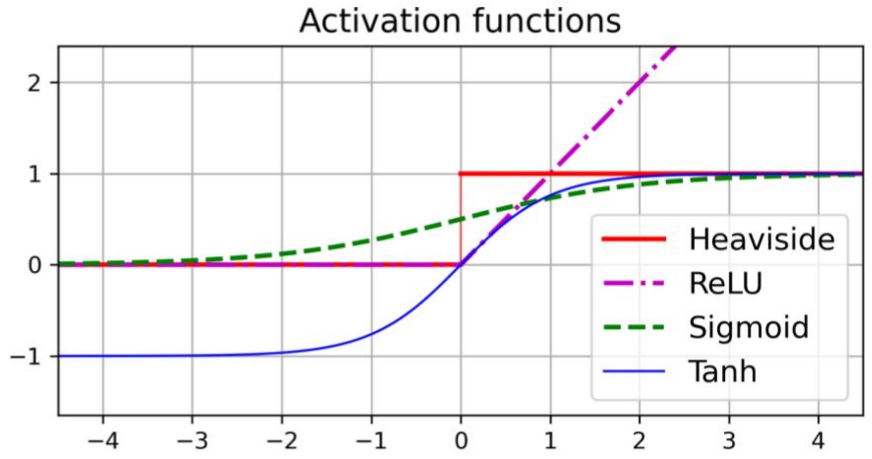
In order for backprop to work properly, Rumelhart and his colleagues made a key change to the MLP’s architecture: they replaced the step function with the logistic function, $\sigma(z) = 1 / (1 + \exp(-z))$, also called the *sigmoid* function. This was essential because the step function contains only flat segments, so there is no gradient to work with (gradient descent cannot move on a flat surface), while the sigmoid function has a well-defined nonzero derivative everywhere, allowing gradient descent to make some progress at every step.

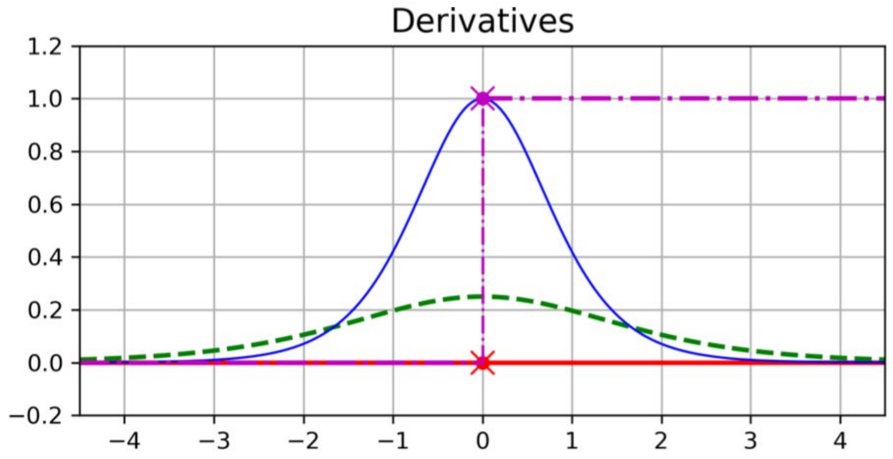
The Multilayer Perceptron and Backpropagation

The backpropagation algorithm works well with many other activation functions, not just the sigmoid function. Here are two other popular choices:

The hyperbolic tangent function: tanh(z) = 2σ(2z) – 1
Just like the sigmoid function, this activation function is *S*-shaped, continuous, and differentiable, but its output value ranges from –1 to 1 (instead of 0 to 1 in the case of the sigmoid function).

The rectified linear unit function: ReLU(z) = max(0, z)
The ReLU function is continuous but unfortunately not differentiable at $z = 0$ (the slope changes abruptly, which can make gradient descent bounce around), and its derivative is 0 for $z < 0$. In practice, however, it works very well and has the advantage of being fast to compute, so it has become the default.





The Multilayer Perceptron and Backpropagation

Why do we need activation functions in the first place?

If you chain several linear transformations, all you get is a linear transformation. For example, if f(x) = 2x + 3 and g(x) = 5x – 1, then chaining these two linear functions gives you another linear function: f(g(x)) = 2(5x – 1) + 3 = 10x + 1. So if you don’t have some nonlinearity between layers, then even a deep stack of layers is equivalent to a single layer, and you can’t solve very complex problems with that. Conversely, a large enough DNN with nonlinear activations can theoretically approximate any continuous function.

Regression MLPs

If you want to predict a single value (e.g., the price of a house, given many of its features), then you just need a single output neuron: its output is the predicted value.

For multivariate regression (i.e., to predict multiple values at once), you need one output neuron per output dimension. For example, to locate the center of an object in an image, you need to predict 2D coordinates, so you need two output neuron

Note that it is important to standardize the input features. This is very important for neural networks because they are trained using gradient descent, and gradient descent does not converge very well when the features have very different scales.

Regression MLPs

Note that this MLP does not use any activation function for the output layer, so it’s free to output any value it wants. This is generally fine, but if you want to guarantee that the output will always be positive, then you should use the ReLU activation function in the output layer.

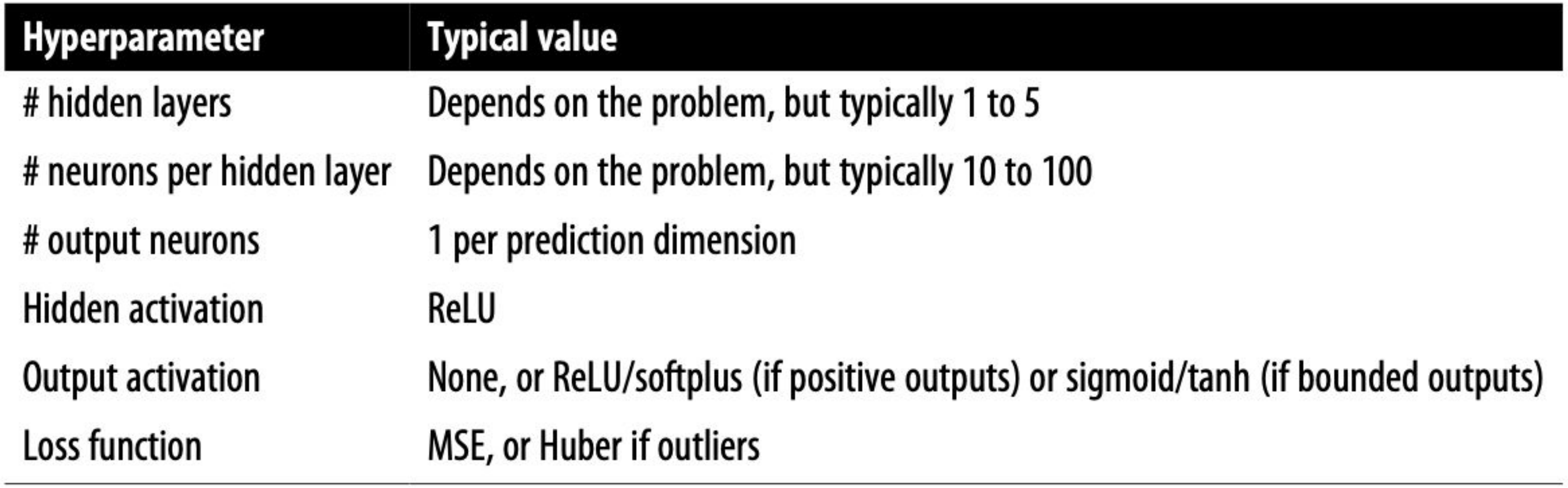
If you want to guarantee that the predictions will always fall within a given range of values, then you should use the sigmoid function or the hyperbolic tangent, and scale the targets to the appropriate range: 0 to 1 for sigmoid and –1 to 1 for tanh.

Regression MLPs

The Regression MLPs typically use the mean squared error, which is usually what you want for regression, but if you have a lot of outliers in the training set, you may prefer to use the mean absolute error instead.

Alternatively, you may want to use the *Huber loss*, which is a combination of both.

Typical Regression MLP Architecture



Hyperparameter	Typical value
# hidden layers	Depends on the problem, but typically 1 to 5
# neurons per hidden layer	Depends on the problem, but typically 10 to 100
# output neurons	1 per prediction dimension
Hidden activation	ReLU
Output activation	None, or ReLU/softplus (if positive outputs) or sigmoid/tanh (if bounded outputs)
Loss function	MSE, or Huber if outliers

Classification MLPs

MLPs can also be used for classification tasks. For a binary classification problem, you just need a single output neuron using the sigmoid activation function: the output will be a number between 0 and 1, which you can interpret as the estimated probability of the positive class. The estimated probability of the negative class is equal to one minus that number.

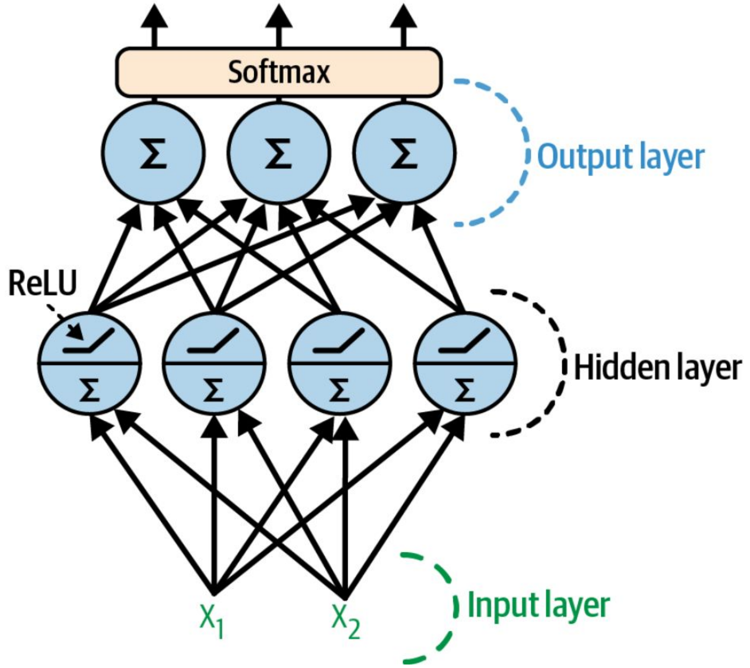
Classification MLPs

MLPs can also easily handle multilabel binary classification tasks. For example, you could have an email classification system that predicts whether each incoming email is ham or spam, and simultaneously predicts whether it is an urgent or nonurgent email. In this case, you would need two output neurons, both using the sigmoid activation function: the first would output the probability that the email is spam, and the second would output the probability that it is urgent. You would dedicate one output neuron for each positive class. Note that the output probabilities do not necessarily add up to 1. This lets the model output any combination of labels: you can have non urgent ham, urgent ham, nonurgent spam, and even urgent spam (although that would probably be an error).

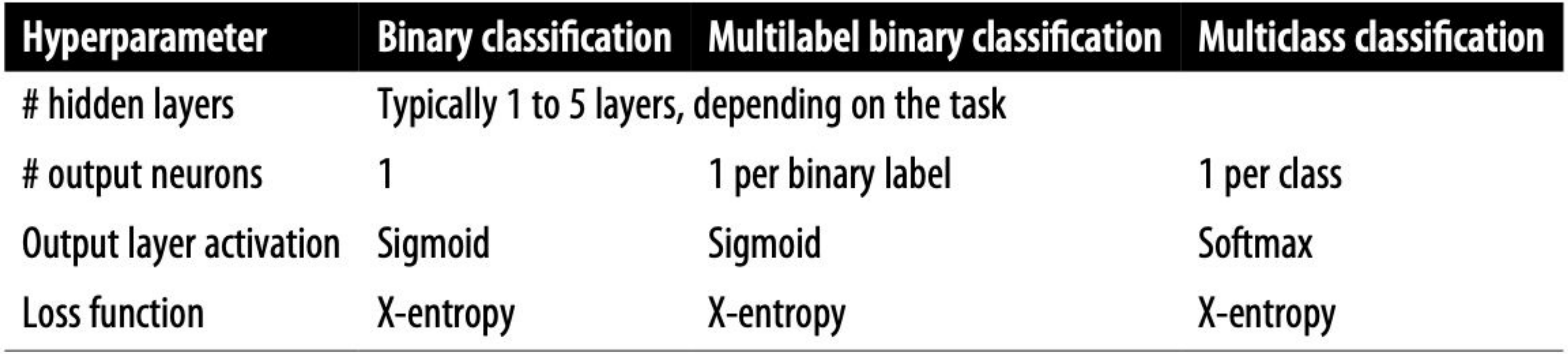
Multi-class Classification

If each instance can belong only to a single class, out of three or more possible classes (e.g., classes 0 through 9 for digit image classification), then you need to have one output neuron per class, and you should use the softmax activation function for the whole output layer. The softmax function will ensure that all the estimated probabilities are between 0 and 1 and that they add up to 1, since the classes are exclusive.

Regarding the loss function, since we are predicting probability distributions, the cross-entropy loss is generally a good choice.



Typical Classification MLP Architecture



Hyperparameter	Binary classification	Multilabel binary classification	Multiclass classification
# hidden layers	Typically 1 to 5 layers, depending on the task		
# output neurons	1	1 per binary label	1 per class
Output layer activation	Sigmoid	Sigmoid	Softmax
Loss function	X-entropy	X-entropy	X-entropy

Implementing MLPs with Keras

Keras is TensorFlow’s high-level deep learning API: it allows you to build, train, evaluate, and execute all sorts of neural networks. The original Keras library was developed by François Chollet as part of a research project and was released as a standalone open source project in March 2015. It quickly gained popularity, owing to its ease of use, flexibility, and beautiful design.

Implementing MLPs with Keras

Keras used to support multiple backends, including TensorFlow, PlaidML, Theano, and Microsoft Cognitive Toolkit (CNTK), but since version 2.4, Keras is TensorFlow-only. Similarly, TensorFlow used to include multiple high-level APIs, but Keras was officially chosen as its preferred high-level API when TensorFlow 2 came out. Installing TensorFlow will automatically install Keras as well, and Keras will not work without TensorFlow installed. Other popular deep learning libraries include PyTorch by Facebook.

Colab runtimes come with recent versions of TensorFlow and Keras preinstalled.

Fine-Tuning Neural Network Hyperparameters

The flexibility of neural networks is also one of their main drawbacks: there are many hyperparameters to tweak. Not only can you use any imaginable network architecture, but even in a basic MLP you can change the number of layers, the number of neurons and the type of activation function to use in each layer, the weight initialization logic, the type of optimizer to use, its learning rate, the batch size, and more.

You can use the *Keras Tuner* library, which is a hyperparameter tuning library for Keras models. It offers several tuning strategies,

Fine-Tuning Neural Network Hyperparameters

Hyperparameter tuning is still an active area of research, and many other approaches are being explored.

But despite all this exciting progress and all these tools and services, it still helps to have an idea of what values are reasonable for each hyperparameter so that you can build a quick prototype and restrict the search space.

Number of Hidden Layers

For many problems, you can begin with a single hidden layer and get reasonable results.

An MLP with just one hidden layer can theoretically model even the most complex functions, provided it has enough neurons.

But for complex problems, deep networks have a much higher *parameter efficiency* than shallow ones: they can model complex functions using exponentially fewer neurons than shallow nets, allowing them to reach much better performance with the same amount of training data.

Lower hidden layers model low-level structures (e.g., line segments of various shapes and orientations), intermediate hidden layers combine these low-level structures to model intermediate-level structures (e.g., squares, circles), and the highest hidden layers and the output layer combine these intermediate structures to model high-level structures (e.g., faces).

Number of Hidden Layers

Not only does this hierarchical architecture help DNNs converge faster to a good solution, but it also improves their ability to generalize to new datasets.

For example, if you have already trained a model to recognize faces in pictures and you now want to train a new neural network to recognize hairstyles, you can kickstart the training by reusing the lower layers of the first network. Instead of randomly initializing the weights and biases of the first few layers of the new neural network, you can initialize them to the values of the weights and biases of the lower layers of the first network.

This way the network will not have to learn from scratch all the low-level structures that occur in most pictures; it will only have to learn the higher-level structures (e.g., hairstyles). This is called *transfer learning*.

Number of Hidden Layers

For many problems you can start with just one or two hidden layers and the neural network will work just fine. For instance, you can easily reach above 97% accuracy on the MNIST dataset using just one hidden layer with a few hundred neurons, and above 98% accuracy using two hidden layers with the same total number of neurons, in roughly the same amount of training time.

For more complex problems, you can ramp up the number of hidden layers until you start overfitting the training set.

Very complex tasks, such as large image classification or speech recognition, typically require networks with dozens of layers or even hundreds, and they need a huge amount of training data. You will rarely have to train such networks from scratch: it is much more common to reuse parts of a pretrained state-of-the-art network that performs a similar task. Training will then be a lot faster and require much less data.

Number of Neurons per Hidden Layer

The number of neurons in the input and output layers is determined by the type of input and output your task requires. For example, the MNIST task requires 28 × 28 = 784 inputs and 10 output neurons.

As for the hidden layers, it used to be common to size them to form a pyramid, with fewer and fewer neurons at each layer - the rationale being that many low-level features can coalesce into far fewer high-level features.

A typical neural network for MNIST might have 3 hidden layers, the first with 300 neurons, the second with 200, and the third with 100. However, this practice has been largely abandoned because it seems that using the same number of neurons in all hidden layers performs just as well in most cases, or even better; plus, there is only one hyperparameter to tune, instead of one per layer.

Depending on the dataset, it can sometimes help to make the first hidden layer bigger than the others.

Number of Neurons per Hidden Layer

Just like the number of layers, you can try increasing the number of neurons gradually until the network starts overfitting.

In general you will get better performance by increasing the number of layers instead of the number of neurons per layer.

Batch Size

The batch size can have a significant impact on your model’s performance and training time. The main benefit of using large batch sizes is that hardware accelerators like GPUs can process them efficiently, so the training algorithm will see more instances per second. Therefore, many researchers and practitioners recommend using the largest batch size that can fit in GPU RAM.

In practice, large batch sizes often lead to training instabilities, especially at the beginning of training, and the resulting model may not generalize as well as a model trained with a small batch size.

Activation Function

In general, the ReLU activation function will be a good default for all hidden layers, but for the output layer it really depends on your task.

Learning Rate, Batch Size, and Other Hyperparameters

The learning rate is arguably the most important hyperparameter.

The optimal learning rate depends on the other hyperparameters - especially the batch size - so if you modify any hyperparameter, make sure to update the learning rate as well.