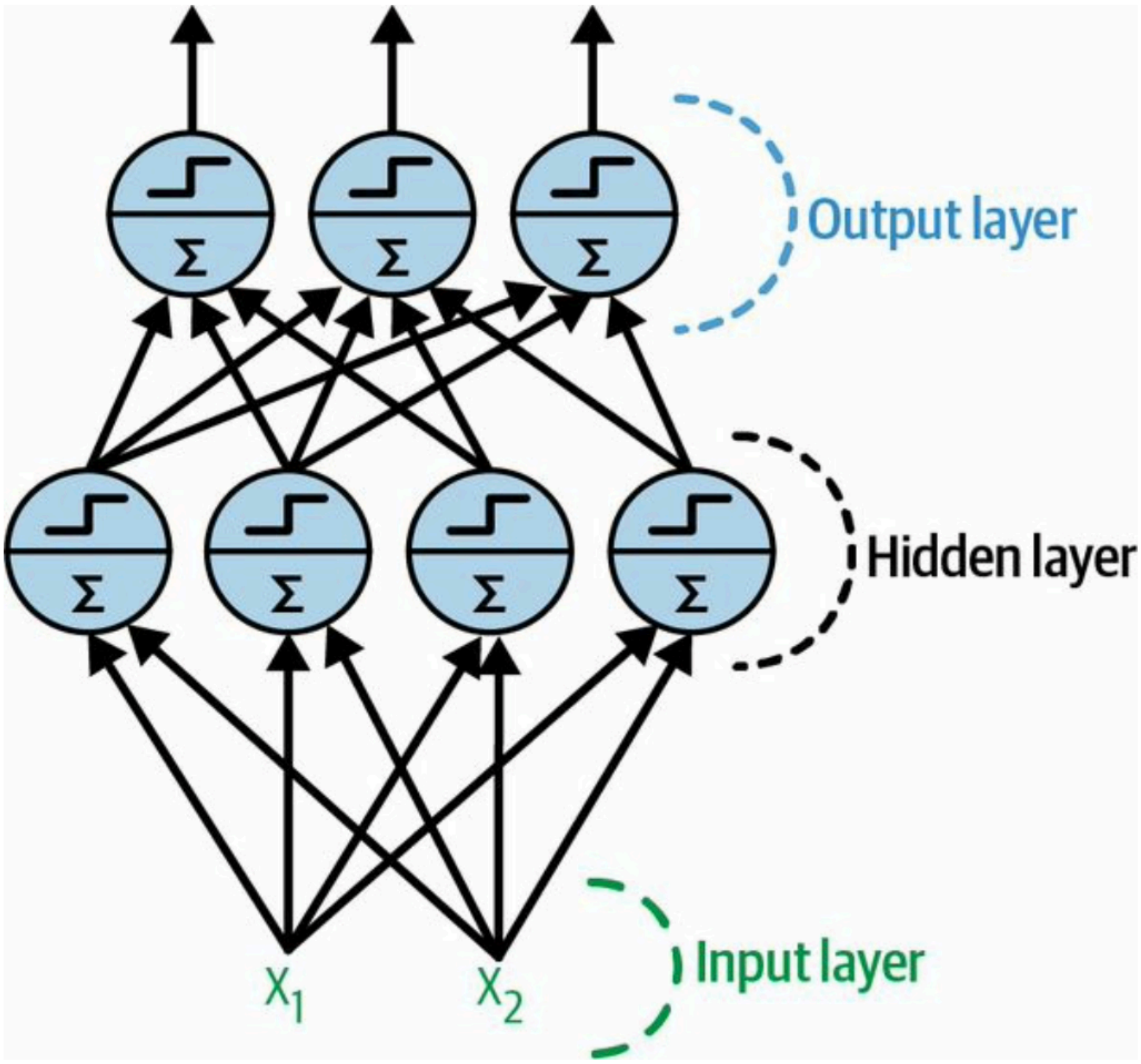
MULTILAYER PERCEPTRON

An MLP is composed of one input layer or more layers of TLU's called hidden layers and one final layer of TLU called the output layer. Layers close to input layers are called lower layers, and ones close to output is called upper layer.



The signal flows only in one direction, so this architecture is an example of feedforward naural network.

When ANN contains a deep stack of hidden layers it is called a deep neural network.
A technique to compute all the gradients automatically and efficiently is an algorithm called reverse mode automatic differentiation. In just 2 passes through the network (forward and backward), it is able to compute the gradients of NN error with regard to every single parameter.
It can find out how each connection weight and bias should be tweaked in order to reduce the NN's error.
These gradients can be used to perform a gradient descent step . If we repeat this process of computing the gradients automatically and taking a gradient descent step the NN's error rate will drop down eventually reaching a minimum. This is called Backpropagation (or backdrop).

Backpropagation can be applied to all sort of computational graphs and not just NN's.

How Backpropagation Works:

1. Mini-Batch Processing

During training, instead of using the entire dataset to update weights, we divide it into smaller groups called mini-batches (e.g., a mini-batch of 32 instances). This approach helps balance between the efficiency of full-batch training and the noise of stochastic training.

2. Epochs

An epoch refers to one complete pass through the entire training dataset. After each epoch, the model weights are updated based on the gradients calculated from each mini-batch.

3. Forward Pass

- **Input Layer:** Each mini-batch is fed into the network starting at the input layer.
- **Activation:** For each neuron in the first hidden layer, the inputs are weighted and summed, then passed through an activation function (e.g., ReLU, sigmoid) to produce the output. This process is repeated for each subsequent layer until we reach the output layer.
- **Intermediate Results:** Unlike standard predictions, during training, we store the outputs of each layer (activations) as they will be needed for the backward pass.

4. Loss Calculation

Once the forward pass is complete, we compute the error of the network's output:

- **Loss Function:** This function quantifies the difference between the predicted outputs (from the output layer) and the actual targets (ground truth). Common loss functions include mean squared error for regression tasks and cross-entropy for classification tasks.

5. Backward Pass (Backpropagation)

Now, we need to calculate how much each weight and bias contributed to the overall error:

- **Error Gradient Calculation:** We begin at the output layer, computing the gradient of the loss with respect to each output neuron's activation. This involves applying the chain rule, which allows us to express the derivative of the loss function in terms of the derivatives of the activation functions used in each layer.
- **Layer-wise Propagation:** With the gradients for the output layer calculated, we propagate these errors backward through the network:
  - For each neuron in the previous layer, we compute how much it contributed to the error of the layer above it. This involves taking the derivative of the activation function for that layer and multiplying it by the error gradient from the layer above.
  - This process continues layer by layer until we reach the input layer, systematically calculating the gradient for each weight and bias.

6. Weight Update

After the gradients have been computed:

- **Gradient Descent Step:** The weights and biases are adjusted using a gradient descent optimization algorithm. For each weight w and bias b, the updates can be formulated

7. Repeat

This process is repeated for all mini-batches in the dataset, and multiple epochs are run until the model converges (i.e., the loss stops improving significantly).

It is important to initialize the hidden layers connection weight randomly or else training will fail.Randomly initializing the connection weights in hidden layers is crucial for effective neural network training. It breaks symmetry, allowing different neurons to learn unique features instead of identical patterns. This diversity helps gradient descent explore the loss landscape more effectively, reducing the risk of poor local minima. Proper initialization also mitigates issues like vanishing and exploding gradients, particularly in deep networks, and can lead to faster convergence.

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).

One key change to the architecture of multilayer perceptrons (MLPs) was replacing the step function with the logistic (sigmoid) function. This modification was essential because the step function has flat segments where the gradient is zero, making it impossible for gradient descent to make updates in those regions. In contrast, the sigmoid function has a nonzero derivative everywhere, enabling gradient descent to make meaningful progress at each step. This change enhances the model's ability to learn from data and improve performance during training.

RELU
The ReLU (Rectified Linear Unit) activation function is defined as:
ReLU(z)=max(0,z).
Continuous but not differentiable at z = 0.

The ReLU activation function is a widely used non-linear activation function.

Output Behavior:

- If the input zzz is greater than zero, ReLU outputs zzz.
- If zzz is less than or equal to zero, ReLU outputs 0.

Activation functions are essential in neural networks because they introduce non-linearity into the model. Without activation functions, chaining multiple linear transformations results in another linear transformation. For example, if you have two linear functions,
f(x)=2x+3f(x)
g(x)=5x−1g(x)
their composition f(g(x)) is still linear: 10x+1This means that no matter how many layers you stack, the overall transformation remains linear.

Regression MLP

MLP can be used for regression tasks . If you want to predict a single value then you need a single output neuron, its output for the predicted value. For multivariate regression, we need one output neuron per output dimension.

Structure of a Regression MLP

1. **Input Layer**:
   - Receives input features (e.g., numerical or categorical data transformed into numerical format).
2. **Hidden Layers**:
   - One or more hidden layers with multiple neurons. Each neuron applies a linear transformation followed by a non-linear activation function (e.g., ReLU, sigmoid) to learn complex patterns.
3. **Output Layer**:
   - For regression tasks, the output layer typically has a single neuron (for single-output regression) with a linear activation function. This allows it to produce a continuous output.

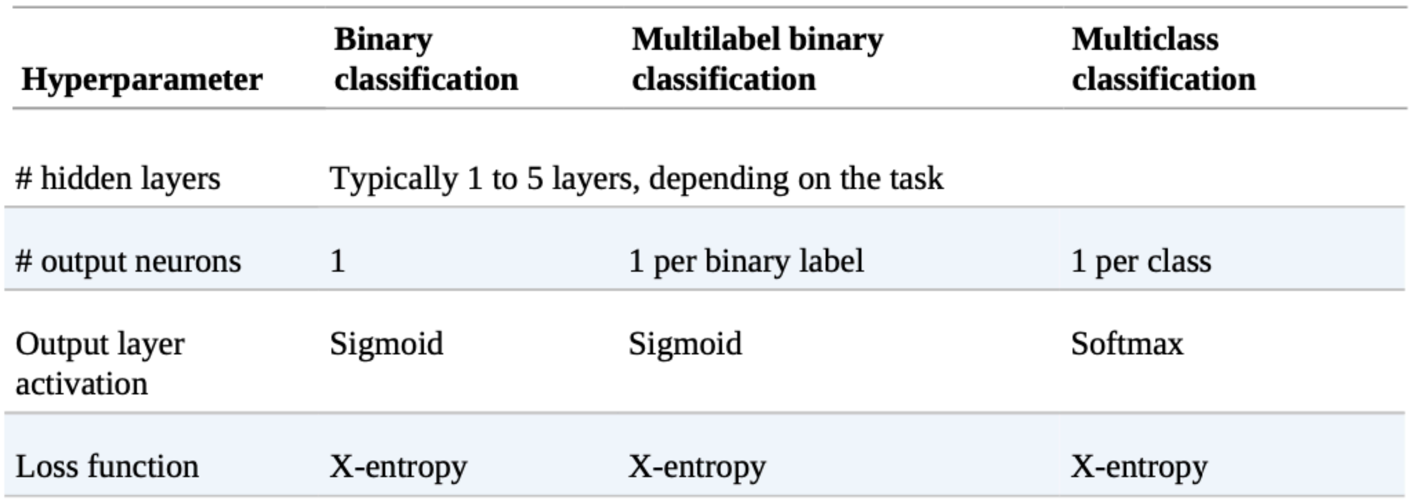
Hyperparameter	**Typical value**
# hidden layers	Depends on the problem, but typically 1 to 5
# neurons per hidden layer	Depends on the problem, but typically 10 to 100
# output neurons	1 per prediction dimension
Hidden activation	ReLU
Output activation	None, or ReLU/softplus (if positive outputs) or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

Classification MLP

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.

MLPs can also easily handle multi label 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 non urgent 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. More generally, 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, non urgent spam, and
perhaps even urgent spam.

For the loss function , since we are predicting the probability distribution, the cross entropy loss or log loss is a good choice.



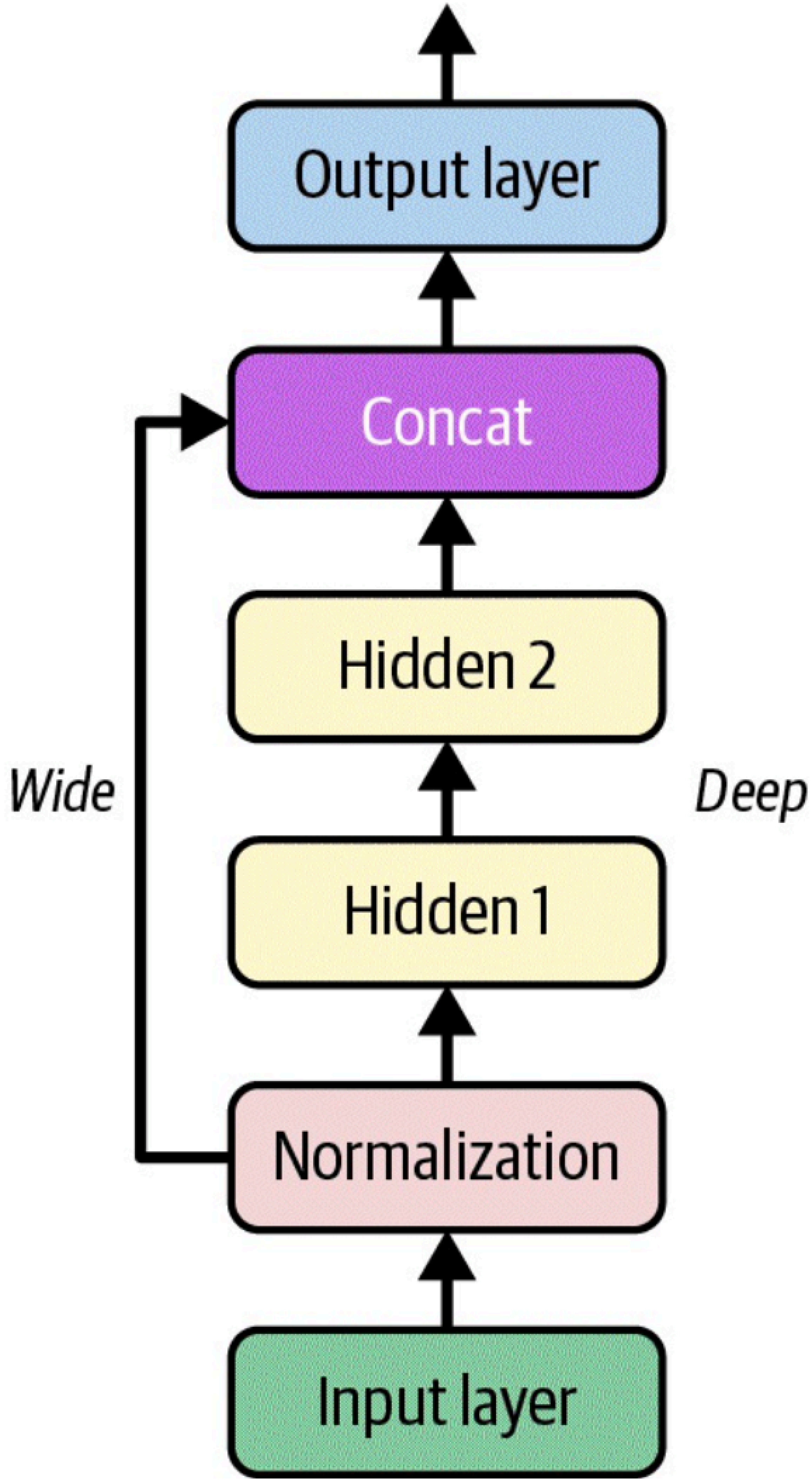
Hyperparameter	Binary classification	Multilabel binary classification	Multiclass classification
# hidden layers	Typically 1 to 5 layers, depending on the task		
# output neurons	1	1 per binary label	1 per class
Output layer activation	Sigmoid	Sigmoid	Softmax
Loss function	X-entropy	X-entropy	X-entropy

A **sequential model** is a type of neural network architecture in which layers are stacked sequentially, meaning that each layer has exactly one input tensor and one output tensor. This is a straightforward way to build a neural network where the flow of data is linear from the input layer through the hidden layers to the output layer.

One example of a nonsequential neural network is a Wide & Deep neural network.

It connects all or part of the inputs directly to the output layer, as shown in Figure 10-13. This architecture makes it possible for the neural network to learn both deep patterns.

In contrast, a regular MLP forces all the data to flow through the full stack of layers; thus, simple patterns in the data may end up being distorted by this sequence of transformations.


Output layer
Concat
Hidden 2
Wide
Deep
Hidden 1
Normalization
Input layer

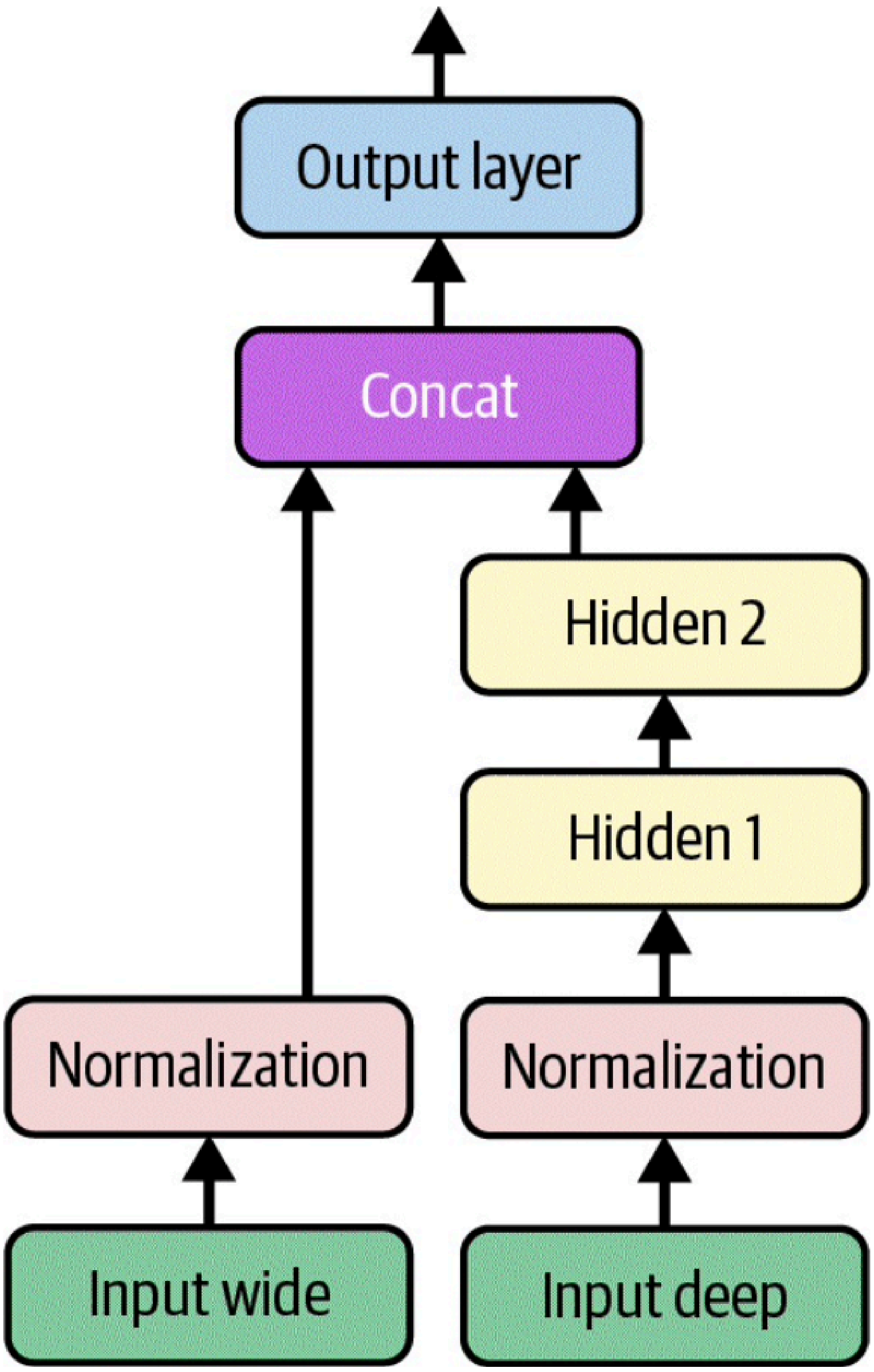


Figure 10-14. Handling multiple inputs

Fine Tuning NN Hyperparameters

The flexibility of neural networks is both a significant strength and a notable challenge. One of the main drawbacks is the sheer number of hyperparameters that need to be tuned. Neural networks allow for a wide range of architectures, including the number of layers and neurons in each layer, the type of activation functions, weight initialization strategies, optimizers, learning rates, batch sizes, and more. This abundance of options creates a complex landscape for model development.

The large number of combinations of hyperparameters results in a vast search space, making it difficult to find the optimal configuration. Hyperparameter tuning often relies on extensive experimentation, which can be both time-consuming and computationally expensive. Additionally, with increased flexibility comes a heightened risk of overfitting, particularly if the model complexity exceeds what the training data can support.

Moreover, there is no one-size-fits-all solution; the optimal settings for hyperparameters can vary significantly depending on the specific dataset and problem at hand, necessitating tailored approaches for each scenario. To navigate this complexity, techniques such as grid search, random search, and more advanced methods like Bayesian optimization or automated hyperparameter tuning tools are often employed. While the flexibility of neural networks allows for powerful modeling capabilities, it requires careful tuning and experimentation to achieve the best results.

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.

Real-world data is often structured hierarchically, and deep neural networks (DNNs) naturally leverage this structure. In these networks, lower hidden layers learn to recognize low-level features, such as lines and edges, while intermediate layers combine these features to identify more complex shapes like squares and circles. The highest layers and the output layer then integrate these intermediate structures to recognize high-level concepts, such as faces. This hierarchical architecture not only accelerates convergence to effective solutions but also enhances the model's ability to generalize to new datasets.

For instance, if you have trained a model to recognize faces and want to develop a new neural network to identify hairstyles, you can benefit from **transfer learning**. Instead of initializing the weights and biases of the lower layers of the new network randomly, you can set them to the values learned by the lower layers of the first network. This approach allows the new model to build upon previously learned low-level structures, so it only needs to focus on learning the higher-level features specific to hairstyles. This method significantly reduces the training time and improves performance on tasks where the underlying features are similar, showcasing the effectiveness of hierarchical learning in DNNs.

For more complex problems, such as large image classification or speech recognition, it's common to increase the number of hidden layers in a neural network until you start to see overfitting on the training set. These tasks often require networks with dozens or even hundreds of layers, although not all layers need to be fully connected, as will be explored in later chapters.

Due to the complexity of these tasks, it's rare to train such deep networks from scratch. Instead, practitioners typically leverage parts of pre trained state-of-the-art networks that are designed for similar tasks. This approach, known as transfer learning, allows you to significantly reduce both training time and the amount of data needed. By starting with a model that has already learned useful feature representations, you can fine-tune it for your specific problem, achieving high performance with much less effort and fewer resources. This strategy is particularly beneficial in fields where obtaining large labeled datasets can be challenging.

Number of neurons per hidden layer

The number of neurons in the input and output layer 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 layer it used to be a common 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.

The practice of varying the number of neurons across different hidden layers has largely fallen out of favor because using a consistent number of neurons in all hidden layers often yields comparable or even better performance. This simplification reduces the complexity of the model, as it limits the number of hyperparameters to tune—just one for the entire network instead of one for each layer.

However, there are exceptions where adjusting the size of the first hidden layer can be beneficial. Depending on the dataset and the specific features being learned, making the first hidden layer larger than subsequent layers may help capture more complex patterns early in the learning process. This approach can be particularly useful in cases where the input data is rich and diverse, allowing the model to extract more relevant features right from the start.

Similar to the approach for determining the number of layers, you can gradually increase the number of neurons in each layer until you observe signs of overfitting. This method allows you to find a suitable balance between model complexity and generalization performance.

Alternatively, another effective strategy is to build a model with a slightly larger architecture—more layers and neurons than you initially believe you'll need. In this case, you can implement techniques such as early stopping and regularization to mitigate overfitting. Early stopping monitors the model's performance on a validation set during training, halting training when performance starts to decline. Regularization techniques, such as dropout or L2 regularization, help prevent the model from becoming too complex by penalizing excessive weights or randomly deactivating neurons during training.

If a layer has too few neurons, it may lack the representational capacity needed to capture and retain all the useful information from the inputs. This limitation can lead to underfitting, where the model fails to learn the underlying patterns in the data. Essentially, with insufficient neurons, the layer cannot effectively transform the input data into a richer representation, resulting in a loss of important features and details.

In general you will get more bang for your buck by increasing the number of layersinstead of the number of neurons per layer.

Learning Rate, Batch Size and Other Hyperparameter

Learning Rate

The learning rate is a crucial hyperparameter in training neural networks, and finding an optimal value is essential for effective learning. A good strategy for determining the optimal learning rate is to start with a very low rate (e.g., 10−510^{-5}10−5) and gradually increase it to a larger value (e.g., 101010) over a set number of iterations (e.g., 500). This can be achieved by multiplying the learning rate by a constant factor at each iteration.

By plotting the loss against the learning rate on a logarithmic scale, you will typically observe a decrease in loss at first, followed by an increase when the learning rate becomes too high. The optimal learning rate is usually a bit lower than the point where the loss begins to rise, often around ten times lower than this turning point. Once the optimal learning rate is identified, you can reinitialize your model and train it normally using this value.

Activation Function

When choosing activation functions for a neural network, the ReLU (Rectified Linear Unit) activation function is generally a good default choice for all hidden layers due to its simplicity and effectiveness in mitigating issues like vanishing gradients. However, the choice of activation function for the output layer depends on the specific task you are addressing.

Iterations

In most cases, the number of training iterations does not actually need to be tweaked: just use early stopping instead.

Batch Size

Batch size can significantly influence both the performance of your model and the time it takes to train it. Large batch sizes allow hardware accelerators like GPUs to process data more efficiently, leading to faster training since the algorithm can handle more instances per second. Many researchers recommend using the largest batch size that fits within the GPU's RAM for optimal performance.

However, there is a potential drawback: large batch sizes can cause training instabilities, particularly during the early stages of training. This instability may result in a model that does not generalize well to new data. One effective strategy is to start with a large batch size in combination with a learning rate warmup, which gradually increases the learning rate during initial training. If you find that training remains unstable or the final performance is unsatisfactory, consider switching to a smaller batch size. This approach allows you to compare results between models trained with different batch sizes, helping you identify the most effective configuration for your specific task.

END