

# **Neural Networks & Deep Learning**

## **[18AI81]**

Teaching Material

Version 0.8

Instructor: Pradip Kumar Das

Companion source code is available at [https://github.com/PradipKumarDas/Teaching/tree/master/18AI81-Neural\\_Networks\\_and\\_Deep\\_Learning/](https://github.com/PradipKumarDas/Teaching/tree/master/18AI81-Neural_Networks_and_Deep_Learning/)

# Table of Contents

|  |    |
|--|----|
| MODULE 1.....  | 4  |
| Introduction to Artificial Neural Networks (ANN).....      | 4  |
| From Biological to Artificial Neurons.....                 | 4  |
| Biological Neurons.....                                    | 4  |
| Logical Computations with Neurons.....                     | 5  |
| The Perceptron.....  | 6  |
| The Multilayer Perceptron and Backpropagation.....         | 9  |
| Regression MLPs.....                                       | 11 |
| Classification MLPs.....                                   | 12 |
| Implementing MLPs with Keras.....                          | 13 |
| Building an Image Classifier Using the Sequential API..... | 13 |
| Building a Regressor MLP Using the Sequential API.....     | 19 |
| Building Complex Models Using Functional API.....          | 20 |
| Saving and Restoring a Model.....                          | 24 |
| Using Callbacks.....                                       | 25 |
| Using TensorBoard for Visualization.....                   | 26 |
| Fine-Tuning Neural Network Hyperparameters.....            | 28 |
| Number of Hidden Layers.....                               | 30 |
| Number of Neurons per Hidden Layer.....                    | 31 |
| Learning Rate, Batch Size, and Other Hyperparameters.....  | 31 |
| Exercises.....   | 32 |
| MODULE 2.....  | 34 |
| Training Deep Neural Networks.....                         | 34 |
| The Vanishing/Exploding Gradient Problems.....             | 34 |
| Glorot and He Initialization.....                          | 35 |
| Better Activation Functions.....                           | 36 |
| Batch Normalization.....                                   | 40 |
| Gradient Clipping.....                                     | 42 |
| Reusing Pretrained Layers.....                             | 43 |
| Transfer Learning with Keras.....                          | 44 |
| Unsupervised Pretraining.....                              | 45 |
| Pretraining on an Auxiliary Task.....                      | 45 |
| Faster Optimizers.....                                     | 46 |
| Momentum.....  | 46 |
| Nesterov Accelerated Gradient.....                         | 47 |
| AdaGrad.....   | 48 |
| RMSProp.....   | 48 |
| Adam.....  | 49 |
| AdaMax.....  | 49 |
| Nadam.....   | 50 |
| AdamW.....   | 50 |
| Avoiding Overfitting Through Regularization.....           | 50 |
| $\ell_1$ and $\ell_2$ Regularization.....                  | 51 |
| Dropout.....   | 51 |
| Monte Carlo Dropout.....                                   | 53 |
| Max-Norm Regularization.....                               | 54 |
| Exercises.....   | 55 |
| MODULE 3.....  | 57 |
| Convolution Neural Networks.....                           | 57 |

|  |     |
|--|-----|
| The Architecture of the Visual Cortex.....                   | 57  |
| Convolution Layers.....                                      | 58  |
| Filters.....   | 60  |
| Stacking Multiple Feature Maps.....                          | 60  |
| Implementing Convolution Layers with Keras.....              | 62  |
| Memory Requirements.....                                     | 62  |
| Pooling Layers.....  | 63  |
| Implementing Pooling Layers with Keras.....                  | 64  |
| CNN Architectures.....                                       | 65  |
| LeNet-5.....   | 67  |
| AlexNet.....   | 67  |
| VGGNet.....  | 70  |
| ResNet.....  | 70  |
| Xception.....  | 72  |
| SENet.....   | 73  |
| Choosing the Right CNN Architecture.....                     | 74  |
| Exercises.....   | 75  |
| Distributing TensorFlow Across Devices and Servers.....      | 77  |
| Serving a TensorFlow Model.....                              | 77  |
| Using TensorFlow Serving.....                                | 77  |
| Using GPUs to Speed Up Computations.....                     | 82  |
| The GPU in the Local Machine.....                            | 83  |
| Managing the GPU RAM.....                                    | 83  |
| Placing Operations and Variables on Devices.....             | 86  |
| Parallel Execution Across Multiple Devices.....              | 86  |
| Training Models Across Multiple Devices.....                 | 88  |
| Model Parallelism.....                                       | 88  |
| Data Parallelism.....  | 90  |
| Training at Scale Using the Distribution Strategies API..... | 94  |
| Training a Model on a TensorFlow Cluster.....                | 95  |
| Exercises.....   | 97  |
| MODULE 4.....  | 98  |
| MODULE 5.....  | 99  |
| REFERENCES.....  | 100 |

# MODULE 1

## Introduction to Artificial Neural Networks (ANN)

It's brain's architecture for inspiration on how to build an intelligent machine. This is the logic that sparked **artificial neural networks (ANNs)**. An ANN is a Machine Learning model inspired by the networks of **biological neurons** found in our brains. But ANNs are **quite different** from their biological cousins.

ANNs are at the very core of **Deep Learning**. They are versatile, powerful, and scalable, making them ideal to tackle **large and highly complex Machine Learning tasks** such as classifying images, speech recognition, recommending products and services, playing games, etc.

### From Biological to Artificial Neurons

ANNs were first introduced back in 1943 by the **neurophysiologist Warren McCulloch and the mathematician Walter Pitts**. In their landmark paper "A Logical Calculus of Ideas Immanent in Nervous Activity," where they presented a simplified computational model of how biological neurons might work together in animal brains to perform complex computations using propositional logic. This was the first artificial neural network architecture. Since then many other architectures have been invented.

ANNs are of interest primarily for the following reasons.

- 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.
- Even if an ANN algorithm gets stuck in **local optima**, in practice it is found to be close to global minima and the model usually works well.
- Research and development on ANNs has been **receiving funding** and is making real progress.

### Biological Neurons

A biological neuron is an unusual-looking cell mostly found in **animal brains**. It's composed of a **cell body containing the nucleus**, many branching extensions called **dendrites**, plus one very long extension called the **axon**. The axon splits off into many branches called **telodendria**, and at the tip of these branches are minuscule structures called **synapses**, which are **connected to the dendrites** or cell bodies of other neurons. Biological neurons produce short **electrical impulses called signals** which travel along the axons and make the synapses **release chemical signals** called neurotransmitters. When a neuron receives a sufficient amount of these neurotransmitters within a few milliseconds, it fires its own electrical impulses.

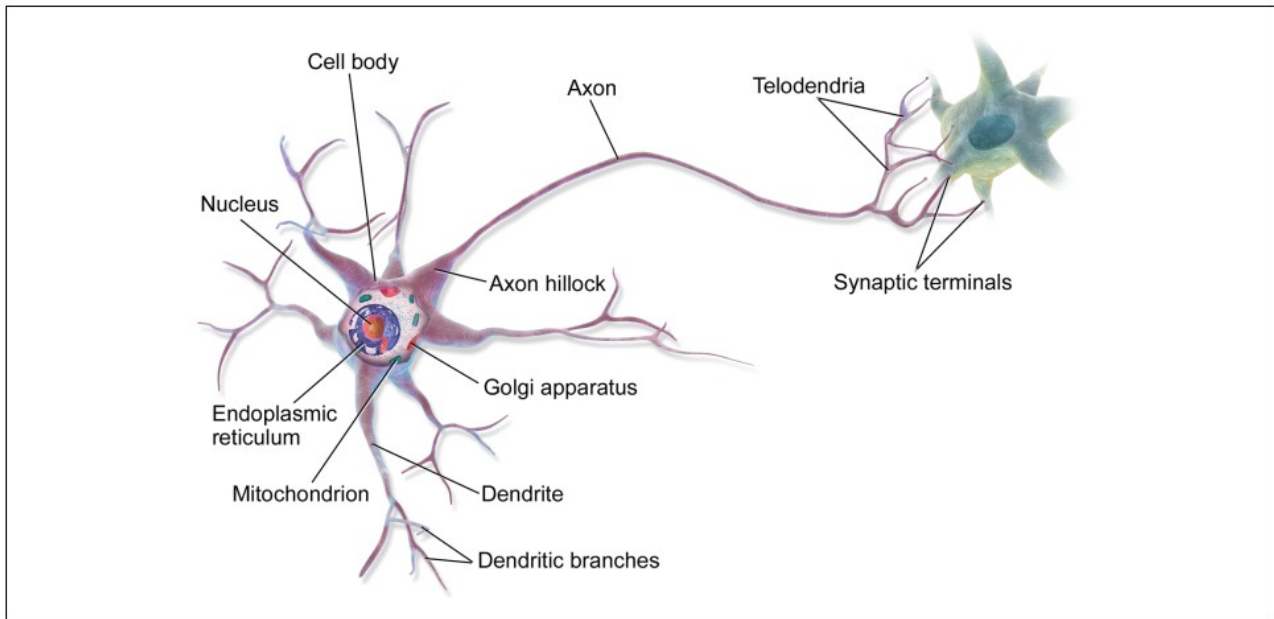


Figure 1: A biological neuron

Though individual biological neurons seem to behave in a rather simple way, they are organized in a vast network of billions, with each neuron typically connected to thousands of other neurons. Highly complex computations can be performed by a network of fairly simple neurons. The architecture of biological neural networks is still the subject of active research, but some parts of the brain have been mapped, and it seems that neurons are often organized in consecutive layers, especially in the cerebral cortex and that has given inspiration to build multilayer neural networks to mimic biological structure of neurons.

### Logical Computations with Neurons

McCulloch and Pitts proposed a very simple model of the biological neuron, which later became known as an artificial neuron: it has one or more binary (on/off) inputs and one binary output. The artificial neuron activates its output when more than a certain number of its inputs are active. Even with such a simplified model it is possible to build a network of artificial neurons that computes any logical proposition. The following ANNs were built to perform various operations as shown below.

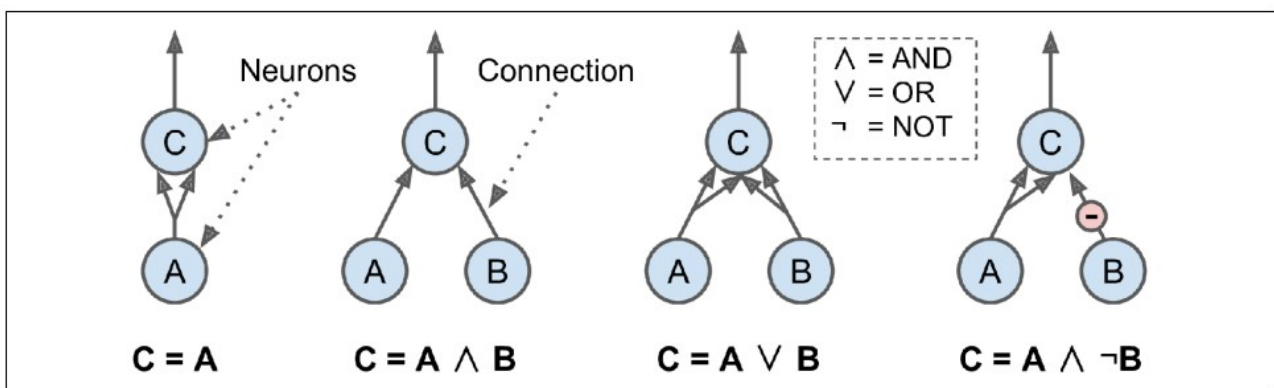


Figure 2: ANNs performing simple logical computations

- The first network on the left is the identity function: if neuron A is activated, then neuron C gets activated as well (since it receives two input signals from neuron A); but if neuron A is off, then neuron C is off as well.
- The second network performs a logical AND: neuron C is activated only when both neurons A and B are activated (a single input signal is not enough to activate neuron C).
- The third network performs a logical OR: neuron C gets activated if either neuron A or neuron B is activated (or both).
- Fourth network computes a slightly more complex logical proposition: neuron C is activated only if neuron A is active and neuron B is off. If neuron A is active all the time, then you get a logical NOT: neuron C is active when neuron B is off, and vice versa.

### 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)**, or sometimes a **linear threshold unit (LTU)**. The inputs and output are numbers (instead of binary on/off values), and each input connection is associated with a weight. The TLU computes a **weighted sum** of its inputs:  $z = w_1x_1 + w_2x_2 + \dots + w_nx_n + b = \mathbf{w}^T \mathbf{x} + b$ . Then applies a **step function** to that sums and outputs the result:  $h_{\mathbf{w},b}(\mathbf{x}) = \text{step}(z)$ . So, it is almost like a **logistic regression**, except it uses a step function instead of the logistic function. Here, the **model parameters are input weights  $\mathbf{w}$  and bias term  $b$** .

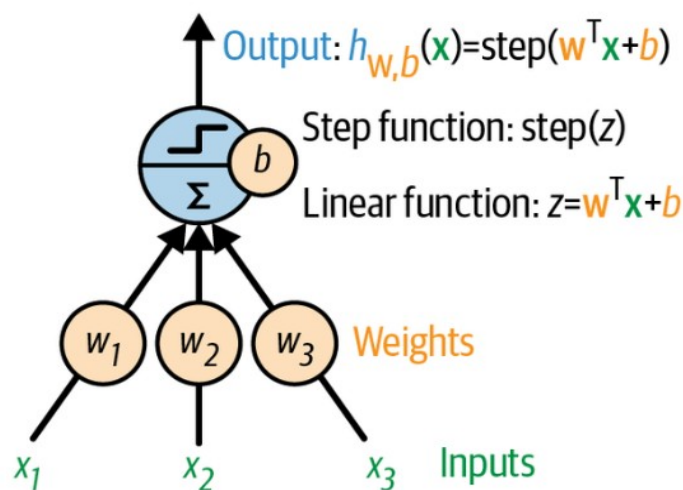


Figure 3: TLU: an artificial neuron that computes a weighted sum of its inputs

The most common step function used in Perceptrons is the **Heaviside step function** and **sign step function**.

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

Equation 1: Common step functions used in perceptrons  
(assuming threshold = 0)

A single TLU can be used for **simple linear binary classification**. It computes a **linear function of its inputs**, and if the result **exceeds a threshold**, it outputs the **positive class**. Otherwise it outputs the **negative class**. For example, a single TLU can be used to classify iris flowers based on petal length and width. Training such a TLU would require finding the right values for  $w_1$ ,  $w_2$ , and  $b$ .

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**.

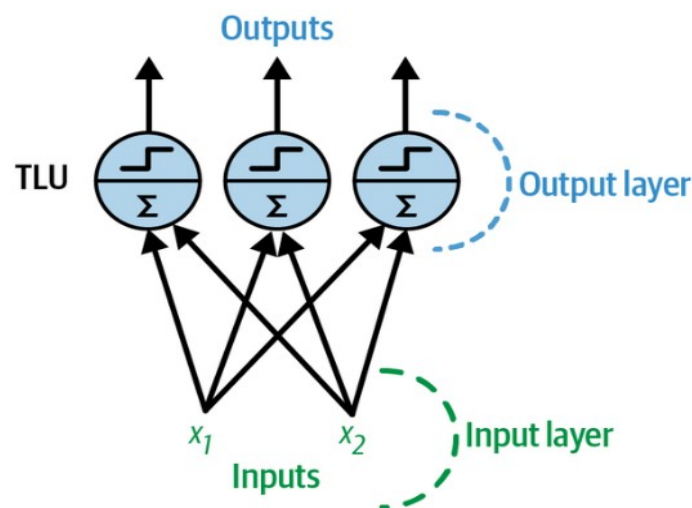


Figure 4: Architecture of a perceptron with two inputs and three output neurons

This perceptron can classify instances simultaneously into **three different binary classes**, which makes it a multilabel classifier. It may also be used for **multiclass classification**. The following equation can be used to efficiently compute the outputs of a layer of artificial neurons for several instances at once.

$$h_{\mathbf{W}}(\mathbf{X}) = \phi(\mathbf{X}\mathbf{W} + \mathbf{b})$$

Equation 2: Computing the outputs of a fully connected layer

In this equation:

- $\mathbf{X}$  represents the **matrix of input features**. It has **one row per instance** and **one column per feature**.
- The **weight matrix**  $\mathbf{W}$  contains all the **connection weights**. It has **one row per input** and **one column per neuron**.

- The **bias vector**  $\mathbf{b}$  contains all the bias terms: **one per neuron**.
- The function  $\phi$  is called the **activation function**: it is a **step function when the artificial neurons are TLUs**.

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**. More specifically, the perceptron is fed one training instance at a time, and for each instance it makes its predictions. 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 **weight update rule** is shown in equation below.

$$w_{ij}^{new} = w_{ij} + \eta (y_j - \hat{y}_j) x_i$$

Equation 3: Perceptron learning rule  
(weight update)

In this equation:

- $w_{ij}$  is the connection weight between the  $i^{\text{th}}$  input and the  $j^{\text{th}}$  neuron.
- $x_i$  is the  $i^{\text{th}}$  input value of the current training instance.
- $y_j$  is the output of the  $j^{\text{th}}$  output neuron for the current training instance.
- $\hat{y}_j$  is the target output of the  $j^{\text{th}}$  output neuron for the current training instance.
- $\eta$  is the learning rate.

The **decision boundary of each output neuron is linear**, so perceptrons are **incapable of learning complex patterns**. However, if the training instances are linearly separable, this algorithm would converge to a solution. This is called the *perceptron convergence theorem*.

Code snippet to classify iris flowers using Perceptron class provided in Scikit-Learn.

```
import numpy as np
from sklearn.datasets import load_iris
from sklearn.linear_model import Perceptron

iris = load_iris(as_frame=True)
X = iris.data[["petal length (cm)", "petal width (cm)"]].values
y = (iris.target == 0) # Iris setosa
per_clf = Perceptron(random_state=42)
per_clf.fit(X, y)

X_new = [[2, 0.5], [3, 1]]
y_pred = per_clf.predict(X_new) # predicts True and False for these 2 flowers
```

Note that Scikit-Learn's Perceptron class is equivalent to the following.

```
SGDClassifier(loss="perceptron", learning_rate="constant",
              eta0=1,           #the learning rate
              penalty=None)     # no regularization
```



But there are a number of **serious weaknesses in perceptrons**—in particular, they are **incapable of solving some trivial problems** such as implementing XOR boolean function. Some of the limitations of perceptrons can be eliminated by stacking multiple perceptrons. The resulting ANN is called a **multilayer perceptron (MLP)**. An MLP can solve the XOR problem as shown in the figure below.

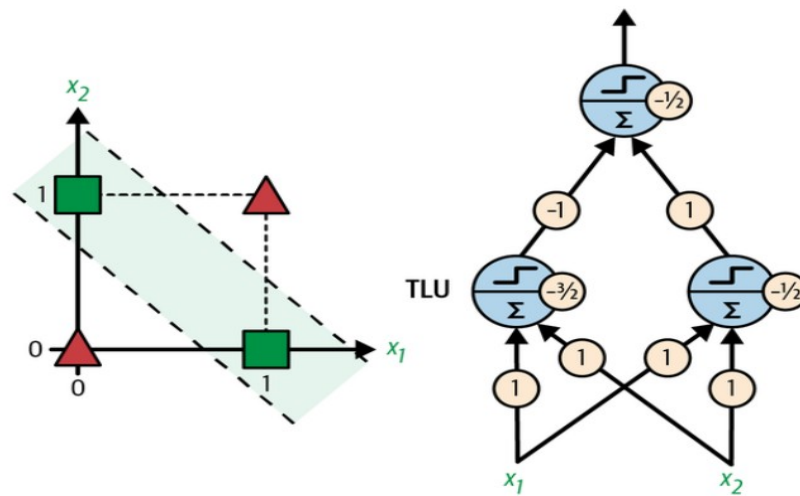


Figure 5: solving XOR function using an MLP

### 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 layers close to the input layer are usually called the **lower layers**, and the ones close to the outputs are usually called the **upper layers**. 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)**.

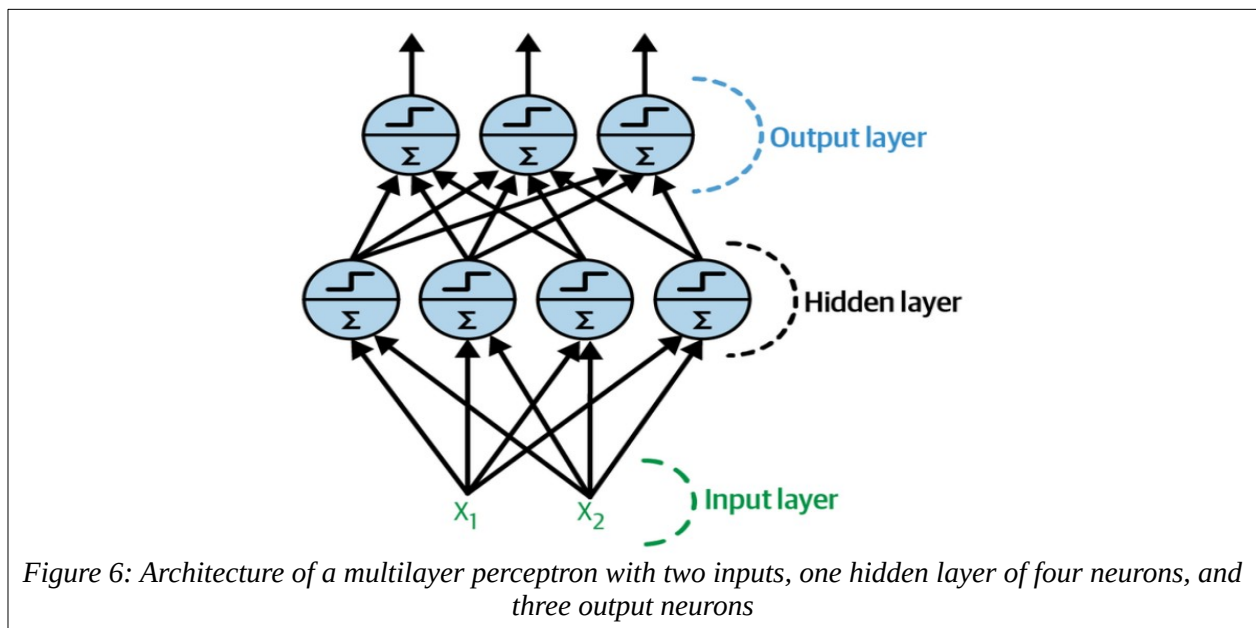


Figure 6: Architecture of a multilayer perceptron with two inputs, one hidden layer of four neurons, and three output neurons

In general, an ANN contains two or more stack of hidden layers, is called a *deep neural network* (DNN). Gradient descent can be used to train neural networks. This requires computing the gradients of the model's error with regard to the model parameters. A technique to compute all the gradients automatically and efficiently is called *reverse-mode automatic differentiation* or *reverse-mode autodiff*. 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. These gradients can then be used to perform a *gradient descent step*. By repeating 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*.

Backpropagation works in the following way.

- It handles one *mini-batch* (by default, containing 32 instances each) at a time, and it goes through the full training set multiple times. Each pass is called an *epoch*.
- 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.
- Next, the algorithm measures the network's output error over a loss function that compares the desired output and the actual output of the network, and returns some measure of the error.
- 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* which makes this step fast and precise.
- 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. 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.
- Finally, the algorithm performs a gradient descent step to tweak all the connection weights in the network, using the error gradients it just computed.

For gradient descent to work with MLP, its architecture was changed by replacing step function a sigmoid function  $\sigma(z) = 1 / (1 + \exp(-z))$ . This was essential because the step function contains only flat segments and 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. In fact, the backpropagation algorithm works well with many other activation functions, not just the sigmoid function. Here are two other popular choices:

**The hyperbolic tangent (or tanh) function:**

$$\tanh(z) = \frac{e^{2z} - 1}{e^{2z} + 1}$$

Equation 4: Tanh activation function

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). That range tends to make each layer's output more or less centered around  $0$  at the beginning of training, which often helps speed up convergence.

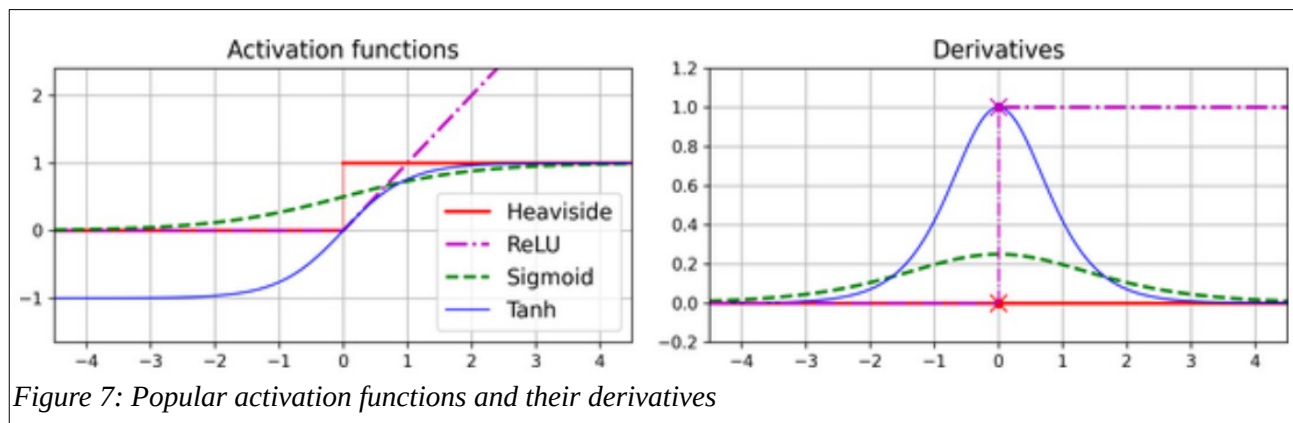
### The rectified linear unit function:

$$\text{ReLU}(z) = \max(0, z)$$

Equation 5: ReLU activation function

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. It does not have a maximum output value.

These popular activation functions and their derivatives are represented in the following figure.



But why are activation functions required in an MLP? It is because chain of linear transformation will result linear transformation. For example, if  $f(x) = 2x + 3$  and  $g(x) = 5x - 1$ , then chaining these two linear functions gives another linear function:  $f(g(x)) = 2(5x - 1) + 3 = 10x + 1$ . So if some nonlinearity between layers does not exist, then even a deep stack of layers is equivalent to a single layer, and very complex problems can not be solved with that. Conversely, a large enough DNN with nonlinear activations can theoretically approximate any continuous function.

### Regression MLPs

Regression MLPs are used for regression tasks. In univariate regression, a single value (e.g., the price of a house, given many of its features) is predicted and that would need a single output neuron. For multivariate regression (i.e., to predict multiple values at once), one output neuron per output dimension would be required.

Scikit-Learn includes an `MLPRegressor` class. The following code builds an MLP with three hidden layers composed of 50 neurons each, and trains it on the California housing dataset. It creates a pipeline to standardize the input features before sending them to the `MLPRegressor`. This is very important for neural networks because they are trained using gradient descent does not converge very well when the features have very different scales. Finally, the code trains the model and evaluates its validation error. The model uses the `ReLU activation function` in the hidden layers, and it uses an optimizer called *Adam* which is a variant of gradient descent, to minimize the mean squared error, with a little bit of  $\ell_2$  regularization.

```
from sklearn.datasets import fetch_california_housing
from sklearn.metrics import mean_squared_error
from sklearn.model_selection import train_test_split
from sklearn.neural_network import MLPRegressor
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import StandardScaler

housing = fetch_california_housing()
X_train_full, X_test, y_train_full, y_test = train_test_split(housing.data,
                                                             housing.target, random_state=42)
X_train, X_valid, y_train, y_valid = train_test_split(
    X_train_full, y_train_full, random_state=42)

mlp_reg = MLPRegressor(hidden_layer_sizes=[50, 50, 50], random_state=42)
pipeline = make_pipeline(StandardScaler(), mlp_reg)
pipeline.fit(X_train, y_train)

y_pred = pipeline.predict(X_valid)
rmse = mean_squared_error(y_valid, y_pred, squared=False) # about 0.505
```

MLP does not use any activation function for the output layer making it free to output any value it computes. If output is required to be positive, then `ReLU activation function` should be used in the output layer, or the `softplus activation function`, which is a smooth variant of ReLU:

$\text{softplus } z = \log(1 + \exp z)$ . Softplus is close to 0 when  $z$  is negative, and close to  $z$  when  $z$  is positive. If the predictions is required to always fall within a given range of values, then `sigmoid function` or the `hyperbolic tangent (tanh) function` should be used to scale the targets to the appropriate range: 0 to 1 for sigmoid and -1 to 1 for tanh. It is to be noted the `MLPRegressor class` does not support activation functions in the output layer.

### Classification MLPs

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

MLPs can also easily handle multi-label binary classification tasks. For example, an email classification system can predict whether each incoming email is ham or spam, and can simultaneously predict whether it is an urgent or non-urgent email. In this case, two output neurons both with sigmoid activation function are required - the first would output the probability that the email is spam, and the second would output the urgency probability.

The **softmax activation function** for the whole output layer should be used when 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 one output neuron per class would be required. 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, the **cross-entropy loss** (or x-entropy or log loss for short) is generally a good choice while predicting probability distributions.

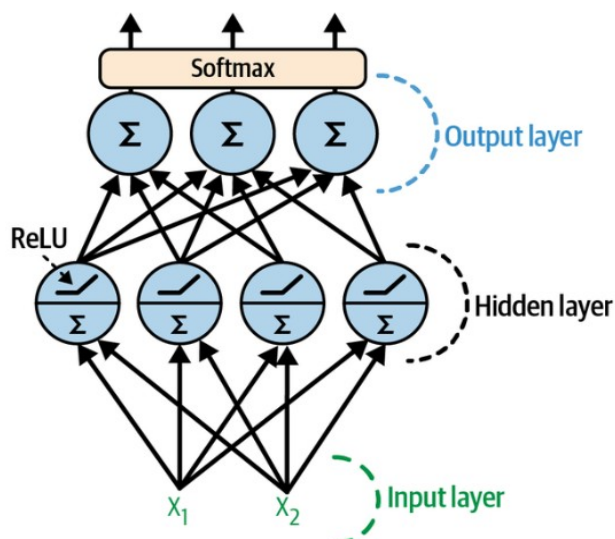


Figure 8: A MLP (with ReLU and softmax) for classification

Scikit-Learn has an **MLPClassifier** class in the `sklearn.neural_network` package. It is almost identical to the `MLPRegressor` class, except that it **minimizes the cross entropy rather than the MSE.**

## Implementing MLPs with Keras

**Keras is TensorFlow's high-level deep learning API:** it allows 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. Keras used to support multiple backends, but since version 2.4, Keras is *TensorFlow*-only. Keras was officially chosen as TensorFlow's preferred high-level API when TensorFlow 2 came out. Installing TensorFlow will automatically install Keras. Other popular deep learning libraries include **PyTorch** by Facebook and **JAX** by Google.

### ***Building an Image Classifier Using the Sequential API***

Keras provides some utility functions to fetch and load common datasets, including **MNIST**, **Fashion MNIST**, and a few more. The following example loads **Fashion MNIST that contains 70,000 grayscale images of  $28 \times 28$  pixels each, with 10 classes.** It's **already shuffled and split** into a training set (60,000 images) and a test set (10,000 images). Last 5,000 images from the training set for validation will be used as a hold out set.

```
import tensorflow as tf

fashion_mnist = tf.keras.datasets.fashion_mnist.load_data()
(X_train_full, y_train_full), (X_test, y_test) = fashion_mnist
X_train, y_train = X_train_full[:-5000], y_train_full[:-5000]
X_valid, y_valid = X_train_full[-5000:], y_train_full[-5000:]

X_train.shape
(55000, 28, 28) # Every image is represented as a 28 × 28 array

X_train.dtype
dtype('uint8') # pixel intensities are represented as integers (from 0 to 255)

# Scales the pixel intensities down to the 0-1 range by dividing them by 255.0
X_train, X_valid, X_test = X_train / 255., X_valid / 255., X_test / 255.

# Stores the class names into a list
class_names = ["T-shirt/top", "Trouser", "Pullover", "Dress", "Coat",
               "Sandal", "Shirt", "Sneaker", "Bag", "Ankle boot"]
```



Figure 9: Samples from Fashion MNIST

## Creating the model using the sequential API

The below code builds a classification MLP with two hidden layers:

```
# Ensures the random weights of the hidden layers and the output layer will be
# the same every time the notebook is run
tf.random.set_seed(42)

# Creates a Sequential model for neural networks composed of a single
# stack of layers connected sequentially. This is called the sequential API.
model = tf.keras.Sequential()

# First layer (an Input layer) gets created and added into the model.
model.add(tf.keras.layers.Input(shape=[28, 28]))

# A flatten layer converts each input image into a 1D array 784 elements from
# 28 x 28 matrix.
model.add(tf.keras.layers.Flatten())

# A Dense hidden layer with 300 neurons with ReLU activation function gets
# added. Each Dense layer manages its own weight matrix, containing
# all the connection weights between the neurons and their inputs. It also
# manages a vector of bias terms (one per neuron).
```

```

model.add(tf.keras.layers.Dense(300, activation="relu"))

# A second Dense hidden layer with 100 neurons with ReLU activation function
# also gets added.

model.add(tf.keras.layers.Dense(100, activation="relu"))

# A Dense output layer with 10 neurons (one per class) with softmax activation
# function gets added because the classes are exclusive.
model.add(tf.keras.layers.Dense(10, activation="softmax"))

```

Instead of adding the layers one by one as done above, it's often more convenient to pass a list of layers when creating the `Sequential` model. Input layer can also be dropped off and instead the `input_shape` can be specified in the first layer.

```

model = tf.keras.Sequential([
    tf.keras.layers.Flatten(input_shape=[28, 28]),
    tf.keras.layers.Dense(300, activation="relu"),
    tf.keras.layers.Dense(100, activation="relu"),
    tf.keras.layers.Dense(10, activation="softmax")
])

```

The `model.summary()` method displays all the model's layers, including each layer's name (which is automatically generated unless you set it when creating the layer), its output shape (None means the batch size can be anything), and its number of parameters. The summary ends with the **total number of parameters**, including **trainable and non-trainable parameters**.

```

>>> model.summary()
Model: "sequential"

```

| Layer (type)              | Output Shape | Param # |
|---------------------------|--------------|---------|
| flatten (Flatten)         | (None, 784)  | 0       |
| dense (Dense)             | (None, 300)  | 235500  |
| dense_1 (Dense)           | (None, 100)  | 30100   |
| dense_2 (Dense)           | (None, 10)   | 1010    |
| Total params: 266,610     |              |         |
| Trainable params: 266,610 |              |         |
| Non-trainable params: 0   |              |         |

## Compiling the model



After a model is created, `compile()` method is called to specify the `loss function` and the `optimizer to use`. Optionally, a list of `extra metrics` to compute during training and evaluation can also be specified.

```
model.compile(loss="sparse_categorical_crossentropy", optimizer="sgd",  
              metrics=["accuracy"])
```

### Selecting a loss function

Loss function "`sparse_categorical_crossentropy`" is used here because `labels are sparsed` (i.e., for each instance, there is just a target class index, from 0 to 9 in this case), and the classes are exclusive. For one target probability per class for each instance (such as one-hot vectors, e.g., [0., 0., 0., 1., 0., 0., 0., 0., 0., 0.] to represent class 3), loss function "`categorical_crossentropy`" should be used. For binary classification or multi-label binary classification, "`sigmoid`" activation function in the output layer instead of the "`softmax`" activation function with "`binary_crossentropy`" loss should be used.

### Optimizer and metrics

Optimizer parameter "`sgd`" indicates this to be stochastic gradient descent that performs backpropagation algorithm i.e., `reverse-mode autodiff plus gradient descent`. And since this is a classifier, metrics parameter is set to "`accuracy`" to measure its accuracy during training and evaluation.

### Training and evaluating the model

To train the model, its `fit()` method is called.

```
history = model.fit(X_train, y_train, epochs=30, ... validation_data=(X_valid,  
y_valid))  
...  
  
Epoch 1/30  
1719/1719 [=====] - 2s 989us/step  
- loss: 0.7220 - sparse_categorical_accuracy: 0.7649  
- val_loss: 0.4959 - val_sparse_categorical_accuracy: 0.8332  
  
Epoch 2/30  
1719/1719 [=====] - 2s 964us/step  
- loss: 0.4825 - sparse_categorical_accuracy: 0.8332  
- val_loss: 0.4567 - val_sparse_categorical_accuracy: 0.8384  
[...]  
  
Epoch 30/30  
1719/1719 [=====] - 2s 963us/step  
- loss: 0.2235 - sparse_categorical_accuracy: 0.9200  
- val_loss: 0.3056 - val_sparse_categorical_accuracy: 0.8894
```

Input features (`X_train`) and target classes (`y_train`) are passed in, as well as the `number of epochs to train`. Default values for epochs is 1 which would not be enough to converge to a good solution. A `validation set` (optional) is also passed in to `measure the loss and the extra metrics on this set at the end of each epoch`, which is very useful to see how well the model really performs. If



the performance on the training set is much better than on the validation set, then the model is probably **overfitting the training set**, or there is a bug, such as a data mismatch between the training set and the validation set.

The **default batch size is 32**, and since the training set has 55,000 images, the model goes through 1,719 batches per epoch: 1,718 of size 32, and 1 of size 24.

### Handling skewed dataset

If the training set is very skewed, with some classes being overrepresented and others underrepresented, it would be useful to set the `class_weight` argument when calling the `fit()` method, to give a larger weight to underrepresented classes and a lower weight to overrepresented classes for Keras to compute the loss accordingly.

For per-instance weights `sample_weight` argument should be set. If both `class_weight` and `sample_weight` are provided, then Keras multiplies them. Per-instance weights could be useful, for example, if some instances were labeled by experts while others were labeled using a crowd-sourcing platform: more weight will be given to the former. Only sample weights without class weights for the validation set can also be provided by adding them as a third item in the `validation_data` tuple.

**Object History returned by `fit()` method** encapsulates model training information including the loss and extra metrics it measured at the end of each epoch on the training set and on the validation set (if any). To get the learning curves, a Pandas DataFrame is created and a plot is drawn as shown below.

```
import matplotlib.pyplot as plt
import pandas as pd

pd.DataFrame(history.history).plot(figsize=(8, 5), xlim=[0, 29], ylim=[0, 1],
    grid=True, xlabel="Epoch", style=["r--", "r--.", "b-", "b-*"])
plt.show()
```

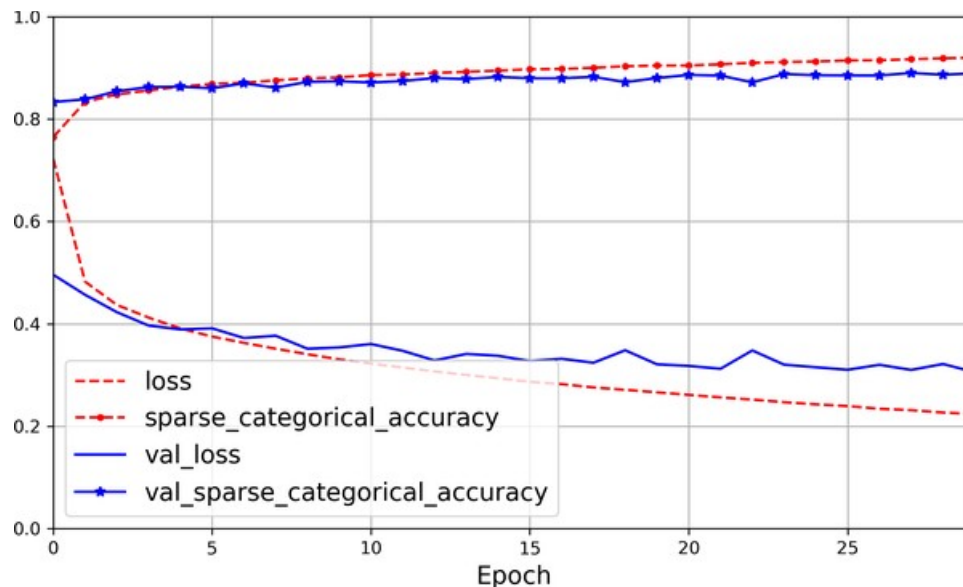


Figure 10: Learning curves: the mean loss and accuracy measured over each epoch, both for training and validation data

As expected, both the training accuracy and the validation accuracy steadily increase during training, while the training loss and the validation loss decrease. The training and validation loss curves are relatively close to each other at first, but they get further apart over time, indicating there's a little bit of overfitting. As the validation loss shows to be going down, the model might not have quite converged yet, so the training should be continuing. This is as simple as calling the `fit()` method again, since Keras just continues training where it left off.

**Hyperparameters** of the model can also be tuned if the model performance is not as expected. **Learning rate** can be the first one for check for. If that doesn't help, a **different optimizer** can be tried. If the performance is still not great, then other model hyperparameters such as the **number of layers**, the **number of neurons per layer**, and the **types of activation functions** to use for each hidden layer can be considered for tuning. **Batch size** can also be considered for tuning by setting new the `batch_size` argument in the `fit()` method. Once model's validation accuracy is at satisfactory level, the model should be evaluated on the test set to estimate the generalization error before the model is deployed to production. This can be done using the `evaluate()` method.

```
model.evaluate(X_test, y_test)

313/313 [=====] - 0s 626us/step
- loss: 0.3243 - sparse_categorical_accuracy: 0.8864
[0.32431697845458984, 0.8863999843597412]
```

## Using the model to make predictions

Model's **`predict()` method** is called to make predictions on new instances. The following code shows the prediction against the first three instances of the test set.

```
X_new = X_test[:3]
y_proba = model.predict(X_new)
y_proba.round(2)
```

```
array([[0. , 0. , 0. , 0. , 0. , 0.01, 0. , 0.02, 0. , 0.97],
       [0. , 0. , 0.99, 0. , 0.01, 0. , 0. , 0. , 0. , 0. ],
       [0. , 1. , 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. ]],
      dtype=float32)
```

For each instance the model estimates one probability per class, from class 0 to class 9. For example, for the first image it estimates that the probability of class 9 (ankle boot) is 97%, the probability of class 7 (sneaker) is 2%, the probability of class 5 (sandal) is 1%, and the probabilities of the other classes are negligible. In other words, it is highly confident that the first image is of an ankle boot. Method `argmax()` method is called to get the highest probability class index for each instance.

```
import numpy as np

y_pred = y_proba.argmax(axis=-1)
y_pred
array([9, 2, 1])

np.array(class_names)[y_pred]
array(['Ankle boot', 'Pullover', 'Trouser'], dtype='<U11')
```

### ***Building a Regressor MLP Using the Sequential API***

The following code uses sequential API to build a model with 3 hidden layers composed of 50 neurons each with a single neuron in the output layer to predict a single value. It uses `no activation function` in the output layer. The `loss function is the mean squared error`, the `metric is the RMSE`. The `optimizer is Adam`. First layer is a `Normalization` layer, but it must be fitted to the training data using its `adapt()` method before calling the model's `fit()` method for this layer to learn the `feature means and standard deviations` in the training data.

```
tf.random.set_seed(42)

norm_layer = tf.keras.layers.Normalization(input_shape=X_train.shape[1:])
model = tf.keras.Sequential([norm_layer,
tf.keras.layers.Dense(50, activation="relu"),
tf.keras.layers.Dense(50, activation="relu"),
tf.keras.layers.Dense(50, activation="relu"),
tf.keras.layers.Dense(1)
])

optimizer = tf.keras.optimizers.Adam(learning_rate=1e-3)
model.compile(loss="mse", optimizer=optimizer, metrics=["RootMeanSquaredError"])
norm_layer.adapt(X_train)

history = model.fit(X_train, y_train, epochs=20,
                    validation_data=(X_valid, y_valid))
mse_test, rmse_test = model.evaluate(X_test, y_test)
X_new = X_test[:3]
y_pred = model.predict(X_new)
```

## Building Complex Models Using Functional API

Though Sequential models are extremely common, it is sometimes useful to build non-sequential neural networks with more complex topologies, or with multiple inputs or outputs. For this purpose, Keras offers the functional API to build such Wide & Deep neural networks. It connects all or part of the inputs directly to the output layer, as shown in the blow figure. This architecture makes it possible for the neural network to learn both deep patterns (using the deep path) and simple rules (through the short path).

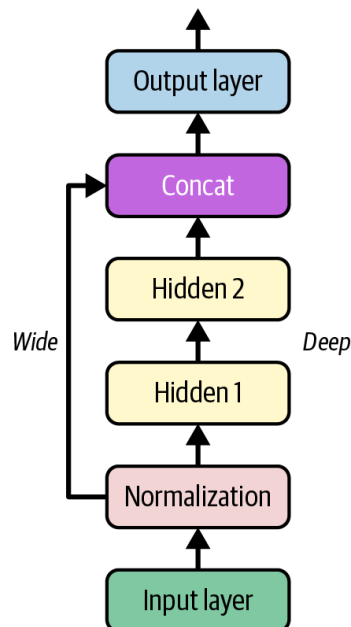


Figure 11: Wide & Deep neural network

The following code builds such a neural network to tackle the California housing problem

```
normalization_layer = tf.keras.layers.Normalization()
hidden_layer1 = tf.keras.layers.Dense(30, activation="relu")
hidden_layer2 = tf.keras.layers.Dense(30, activation="relu")
concat_layer = tf.keras.layers.Concatenate()
output_layer = tf.keras.layers.Dense(1)

input_ = tf.keras.layers.Input(shape=X_train.shape[1:])
normalized = normalization_layer(input_)
hidden1 = hidden_layer1(normalized)
hidden2 = hidden_layer2(hidden1)
concat = concat_layer([normalized, hidden2])
output = output_layer(concat)

model = tf.keras.Model(inputs=[input_], outputs=[output])
```

At a high level, the first five lines create all the layers we need to build the model, the next six lines use these layers just like functions to go from the input to the output, and the last line creates a Keras Model object by pointing to the input and the output.

It is to be noted here that `concat_layer` concatenate the input and the second hidden layer's output. Other steps such as compiling the model, adapting the Normalization layer, fitting the model, evaluating it, and using it to make predictions remain the same as that of sequential API.

## Handling Multiple Inputs

Possible approach to send a subset of the features through the wide path and a different subset through the deep path is shown both through the figure and source code below.

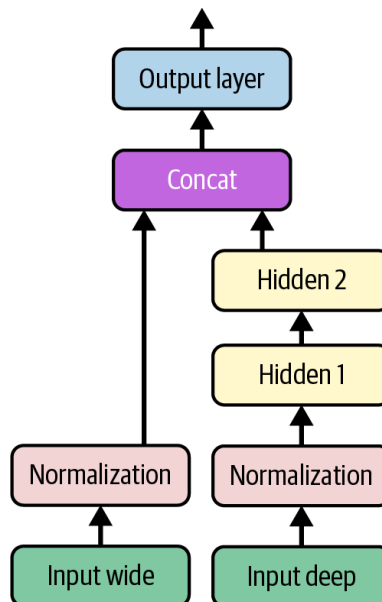


Figure 12: Handling multiple inputs

```
input_wide = tf.keras.layers.Input(shape=[5]) # features 0 to 4
input_deep = tf.keras.layers.Input(shape=[6]) # features 2 to 7

norm_layer_wide = tf.keras.layers.Normalization()
norm_layer_deep = tf.keras.layers.Normalization()
norm_wide = norm_layer_wide(input_wide)
norm_deep = norm_layer_deep(input_deep)

hidden1 = tf.keras.layers.Dense(30, activation="relu")(norm_deep)
hidden2 = tf.keras.layers.Dense(30, activation="relu")(hidden1)

concat = tf.keras.layers.concatenate([norm_wide, hidden2])

output = tf.keras.layers.Dense(1)(concat)

model = tf.keras.Model(inputs=[input_wide, input_deep], outputs=[output])
```

Compilation of the model goes usual, but when calling the `fit()` method, instead of passing a single input matrix `X_train`, a pair of matrices (`X_train_wide`, `X_train_deep`) is passed, one per input. The same is true for `X_valid`, and also for `X_test` and `X_new` when `evaluate()` or `predict()` is called.

```
optimizer = tf.keras.optimizers.Adam(learning_rate=1e-3)
```

```

model.compile(loss="mse", optimizer=optimizer, metrics=["RootMeanSquaredError"])

X_train_wide, X_train_deep = X_train[:, :5], X_train[:, 2:]
X_valid_wide, X_valid_deep = X_valid[:, :5], X_valid[:, 2:]
X_test_wide, X_test_deep = X_test[:, :5], X_test[:, 2:]
X_new_wide, X_new_deep = X_test_wide[:3], X_test_deep[:3]

norm_layer_wide.adapt(X_train_wide)
norm_layer_deep.adapt(X_train_deep)

history = model.fit((X_train_wide, X_train_deep), y_train, epochs=20,
                    validation_data=((X_valid_wide, X_valid_deep), y_valid))
mse_test = model.evaluate((X_test_wide, X_test_deep), y_test)
y_pred = model.predict((X_new_wide, X_new_deep))

```

## Handling Multiple Outputs

There could be many use cases that would require to have **multiple outputs** from the model. For example, locating an **object and classifying it at the same time would require two outputs**. Multiple outputs from multiple neural network to handle independent tasks could also be the requirement.

Example of multi-output model with one auxiliary output is shown below.

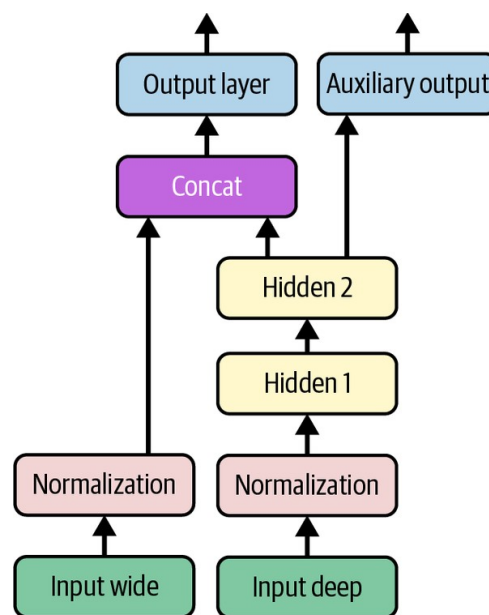


Figure 13: Handling multiple outputs

```

[...] # Same as above, up to the main output layer
output = tf.keras.layers.Dense(1)(concat)
aux_output = tf.keras.layers.Dense(1)(hidden2)

model = tf.keras.Model(inputs=[input_wide, input_deep],
                      outputs=[output, aux_output])

optimizer = tf.keras.optimizers.Adam(learning_rate=1e-3)

model.compile(

```

```

    loss=("mse", "mse"), # Loss functions for each output
    loss_weights=(0.9, 0.1), # More weight for main output loss
    optimizer=optimizer,
    metrics=["RootMeanSquaredError"])

norm_layer_wide.adapt(X_train_wide)
norm_layer_deep.adapt(X_train_deep)

# Provide labels for both outputs for both training and validation set
history = model.fit(
    (X_train_wide, X_train_deep), (y_train, y_train), epochs=20,
    validation_data=((X_valid_wide, X_valid_deep), (y_valid, y_valid))
)

# Returns weighted sum of the losses as well as individual losses and metrics
eval_results = model.evaluate((X_test_wide, X_test_deep), (y_test, y_test))
weighted_sum_of_losses, main_loss, aux_loss, main_rmse, aux_rmse = eval_results

# Return predictions for each output
y_pred_main, y_pred_aux = model.predict((X_new_wide, X_new_deep))

```

## Using the Subclassing API to Build Dynamic Models

Both the sequential API and the functional API are declarative in the sense that layers and their connections can be declared, followed by feeding the model some data for training or inference. This has many advantages including

- the model can easily be saved, cloned, and shared;
- its structure can be displayed and analyzed;
- the framework can infer shapes and check types.

Since the whole model is a static graph of layers, it helps in debugging. But the architecture of the model built with sequential or function API is static. For a more imperative programming style requiring models to have loops, varying shapes, conditional branching and other dynamic behaviors, the subclassing API can be used. With this approach, the `Model` class is subclassed, layers are created in the constructor, and `call()` method is called to perform required computations.

```

class WideAndDeepModel(tf.keras.Model):
    def __init__(self, units=30, activation="relu", **kwargs):
        super().__init__(**kwargs) # needed to support naming the model
        self.norm_layer_wide = tf.keras.layers.Normalization()
        self.norm_layer_deep = tf.keras.layers.Normalization()
        self.hidden1 = tf.keras.layers.Dense(units, activation=activation)
        self.hidden2 = tf.keras.layers.Dense(units, activation=activation)
        self.main_output = tf.keras.layers.Dense(1)
        self.aux_output = tf.keras.layers.Dense(1)

    def call(self, inputs):
        input_wide, input_deep = inputs
        norm_wide = self.norm_layer_wide(input_wide)
        norm_deep = self.norm_layer_deep(input_deep)
        hidden1 = self.hidden1(norm_deep)
        hidden2 = self.hidden2(hidden1)

```

```

concat = tf.keras.layers.concatenate([norm_wide, hidden2])
output = self.main_output(concat)
aux_output = self.aux_output(hidden2)
return output, aux_output

model = WideAndDeepModel(30, activation="relu", name="my_cool_model")

```

Now the model instance can be compiled, adapted to its normalization layers, fitted, evaluated, and used to make predictions, just the way these are done with model created by sequential or functional API. But as model's architecture is hidden within the `call()` method,

- the model cannot be cloned using `tf.keras.models.clone_model()`;
- `summary()` method returns only list of layers without any information on how they are connected to each other;
- Keras cannot check types and shapes ahead of time;

So unless extra flexibility are really needed, the sequential API or the functional API should normally be preferred.

### ***Saving and Restoring a Model***

A trained Keras model can be saved as follows.

```

model.save(<file name>, save_format="tf")

```

TensorFlow's **SavedModel** format is denoted as **"tf"** and the model files get stored in a directory with name given in the first parameter. Important files and folder are mentioned below.

- **Model's architecture and logic** are stored into a file with name **saved\_model.pb**.
- The file **keras\_metadata.pb** contains extra information needed by Keras.
- The **variables** subdirectory contains all the **parameter values including the connection weights, the biases, the normalization statistics, and the parameters of the optimizer**.
- The **assets** directory may contain extra files, such as **data samples, feature names and class names**.

Since the **optimizer is also saved, including its hyperparameters and any state it may have**, after loading the model, the training can be continued, if required.

If model is saved using **save\_format="h5"** or use a filename that ends with **.h5**, **.hdf5**, or **.keras**, then Keras will save the model to a single file using a Keras-specific format based on the **HDF5** format. However, most TensorFlow deployment tools require the **SavedModel** format instead.

A saved model can be loaded as shown below for making evaluation or predictions.

```

model = tf.keras.models.load_model(<file name>)
y_pred_main, y_pred_aux = model.predict((X_new_wide, X_new_deep))

```



Instead of saving the whole model, method `model.save_weights()` and `model.load_weights()` can be called to save and load only the parameter values, respectively. This includes the connection weights, biases, preprocessing statistics, optimizer state, etc. Saving just the weights is faster and uses less disk space than saving the whole model, so it's perfect to save quick checkpoints during training as described in the next section.

### Using Callbacks

The `fit()` method accepts a `callbacks` argument that accepts a list of objects that Keras will call at certain events such as before and after training, before and after each epoch, and even before and after processing each batch. For example, the `ModelCheckpoint` callback saves checkpoints of your model at regular intervals during training, by default at the end of each epoch.

```
checkpoint_cb = tf.keras.callbacks.ModelCheckpoint("my_checkpoints",
    save_weights_only=True)

history = model.fit(..., callbacks=[checkpoint_cb])
```

When creating the `ModelCheckpoint`, parameter `save_best_only=True` can also be set if a validation set is used during training. In this case, it will only save the model when its performance on the validation set is the best so far. The best model on the validation set will be restored after training.

When model training measures no progress on the validation set for a number of epochs, the training can be interrupted by using the `EarlyStopping` callback. It will roll back to the best model at the end of training if parameter `restore_best_weights=True` is set.

Callback to checkpoints will save the model to avoid wasting time and resources in case the computer crashes, and callback to `EarlyStopping` will interrupt training early when there is no more progress to reduce overfitting.

```
early_stopping_cb = tf.keras.callbacks.EarlyStopping(patience=10,
    restore_best_weights=True)

history = model.fit(..., callbacks=[checkpoint_cb, early_stopping_cb])
```

Ensuring the learning rate is not too small, the number of epochs can be set to a large value since training will stop automatically when there is no more progress.

### Custom callbacks

For extra control, custom callbacks can also be written. For example, the following custom callback will display the ratio between the validation loss and the training loss during training to detect overfitting.

```
class PrintValTrainRatioCallback(tf.keras.callbacks.Callback):
    def on_epoch_end(self, epoch, logs):
        ratio = logs["val_loss"] / logs["loss"]
        print(f"Epoch={epoch}, val/train={ratio:.2f}")
```

Custom callbacks can also be used during evaluation and predictions, if required. The following methods are available to be overridden for customization.

| For training   | For evaluation   | For prediction   |
|--|--|--|
| <code>on_train_begin()</code><br><code>on_train_end()</code><br><code>on_epoch_begin()</code><br><code>on_epoch_end()</code><br><code>on_batch_begin()</code><br><code>on_batch_end()</code> | <code>on_test_begin()</code><br><code>on_test_end()</code><br><code>on_test_batch_begin()</code><br><code>on_test_batch_end()</code> | <code>on_predict_begin()</code><br><code>on_predict_end()</code><br><code>on_predict_batch_begin()</code><br><code>on_predict_batch_end()</code> |

### Using TensorBoard for Visualization

TensorBoard is a great interactive visualization tool that can be used to

- view the learning curves during training,
- compare curves and metrics between multiple runs,
- visualize the computation graph,
- analyze training statistics,
- view images generated by your model,
- visualize complex multidimensional data projected down to 3D and get these clustered automatically,
- profile network to measure its speed to identify bottlenecks, and more.

Required modifications need to be made into the program so that it outputs the data in a special binary log files called event files that the TensorBoard will require for visualization. Each binary data record is called a summary. The TensorBoard server will monitor the log directory, and it will automatically pick up the changes and automatically update the visualizations such as the learning curves during training. In general, TensorBoard server is configured to point to a root log directory and program are also configured to write to a different subdirectory every time it runs. This way, the same TensorBoard server instance will allow to visualize and compare data from multiple runs of the program, without getting everything mixed up.

Keras provides a `TensorBoard()` callback that will create the log directory and it will create event files and write summaries to them during training to measure model's training and validation loss and metrics and it will also profile neural network. Parameter `profile_batch` is optional. In the below example, it will profile the network between batches 100 and 200 during the first epoch as it often takes a few batches for the neural network to "warm up".

```
tensorboard_cb = tf.keras.callbacks.TensorBoard(<log directory>,
    profile_batch=(100, 200))
history = model.fit(..., callbacks=[tensorboard_cb])
```

Running the code again with different hyperparameter(s) will create with a new log subdirectory, provided parameter `log_dir` is set accordingly. The directory structure could be similar to the below one. There's one directory per run, each containing one subdirectory for training logs and one for validation logs. Both contain event files, and the training logs and also include profiling traces.

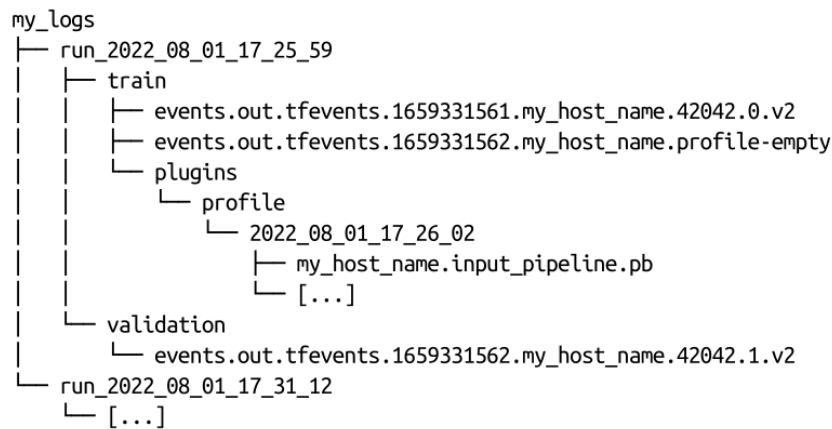


Figure 14: An example directory structure created by TensorBoard

The TensorBoard server can be started directly within Jupyter or Colab using the Jupyter extension for TensorBoard. The following code loads the Jupyter extension for TensorBoard, and the second line starts a TensorBoard server for the `my_logs` directory, connects to this server and displays the user interface directly inside of Jupyter. The server, listens on the `first available TCP port greater than or equal to 6006`. Custom port can be set using the `--port` option.

```
%load_ext tensorboard
%tensorboard --logdir=./my_logs
```

For local development TensorBoard can be started by `executing tensorboard --logdir=./my_logs` in a terminal followed by `opening http://localhost:6006` in the browser to open TensorBoard user interface that looks like the one below.

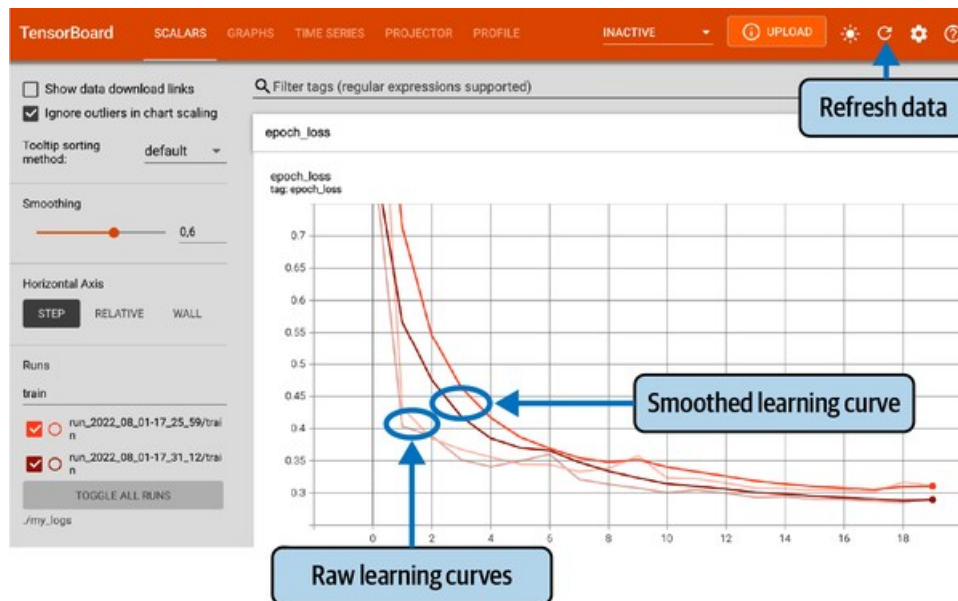


Figure 15: Visualizing learning curves with TensorBoard

Additionally, TensorFlow offers a lower-level API in the `tf.summary` package. First `SummaryWriter` is created using the `create_file_writer()` function, and then it uses this writer as a Python context to `log scalars, histograms, images, audio, and text`, all of which can then be visualized using TensorBoard.

## Fine-Tuning Neural Network Hyperparameters

In a neural network there are `many hyperparameters to tweak`. Even in a basic MLP changes could be done on 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`. Fine-tuning neural network hyperparameters is all about `finding combination of hyperparameters that are the best for the task in hand`.

One of the better ways is to use the `Keras Tuner` library, which is a hyperparameter tuning library for Keras models. It offers several tuning strategies and it is highly customizable, and it has excellent integration with TensorBoard. Tuning is all about writing a function that builds, compiles, and returns a Keras model. This function must take a `kt.HyperParameters` object as an argument, which it can use to define hyperparameters (integers, floats, strings, etc.) along with their `range of possible values`, and these hyperparameters may be used to build and compile the model. For example, the following function builds and compiles an MLP to classify Fashion MNIST images, using hyperparameters such as the number of hidden layers, the number of neurons per layer, the learning rate, and the type of optimizer to use.

```
import keras_tuner as kt

def build_model(hp):
    n_hidden = hp.Int("n_hidden", min_value=0, max_value=8, default=2)
    n_neurons = hp.Int("n_neurons", min_value=16, max_value=256)
    learning_rate = hp.Float("learning_rate", min_value=1e-4, max_value=1e-2,
```

```

        sampling="log")
optimizer = hp.Choice("optimizer", values=["sgd", "adam"])
if optimizer == "sgd":
    optimizer = tf.keras.optimizers.SGD(learning_rate=learning_rate)
else:
    optimizer = tf.keras.optimizers.Adam(learning_rate=learning_rate)

model = tf.keras.Sequential()
model.add(tf.keras.layers.Flatten())

for _ in range(n_hidden):
    model.add(tf.keras.layers.Dense(n_neurons, activation="relu"))

model.add(tf.keras.layers.Dense(10, activation="softmax"))
model.compile(loss="sparse_categorical_crossentropy",
              optimizer=optimizer, metrics=["accuracy"])

return model

```

Now, to do a basic random search, a `kt.RandomSearch` tuner is created, passing the `build_model` function to the constructor, and call the tuner's `search()` method.

```

random_search_tuner = kt.RandomSearch(
    build_model, objective="val_accuracy", max_trials=5, overwrite=True,
    directory="my_fashion_mnist", project_name="my_rnd_search", seed=42)
random_search_tuner.search(X_train, y_train, epochs=10,
    validation_data=(X_valid, y_valid))

```

The `RandomSearch` tuner first calls `build_model()` once with an empty `Hyperparameters` object, just to gather all the hyperparameter specifications. Then, it runs 5 trials; for each trial it builds a model using hyperparameters sampled randomly within their respective ranges, then it trains that model for 10 epochs and saves it to a subdirectory of the `my_fashion_mnist/my_rnd_search` directory. Since `overwrite=True`, the `my_rnd_search` directory is deleted before training starts. If this code is run for a second time but with `overwrite=False` and `max_trials=10`, the tuner will continue tuning where it left off, running 5 more trials meaning all the trials need not be run in one shot. Lastly, since objective is set to "val\_accuracy", the tuner prefers models with a higher validation accuracy, so once the tuner has finished searching, the best models is received as follows.

```

top3_models = random_search_tuner.get_best_models(num_models=3)
best_model = top3_models[0]

```

Method `get_best_hyperparameters()` can be called to get the `kt.HyperParameters` of the best models.

```

top3_params = random_search_tuner.get_best_hyperparameters(num_trials=3)
top3_params[0].values # best hyperparameter values

{'n_hidden': 5,
'n_neurons': 70,
'learning_rate': 0.00041268008323824807,

```

```
'optimizer': 'adam'}
```

All the metrics can also be accessed directly. For example, this shows the best validation accuracy.

```
best_trial.metrics.get_last_value("val_accuracy")  
0.8736000061035156
```

Upon receiving satisfactory performance from the best model, training can be continued for a few epochs on the full training set and then the model can be evaluated on the test set followed by deploying it to production.

```
best_model.fit(X_train_full, y_train_full, epochs=10)  
test_loss, test_accuracy = best_model.evaluate(X_test, y_test)
```

The following sections provide guidelines for choosing the number of hidden layers and neurons in an MLP and for selecting good values for some of the main hyperparameters.

### ***Number of Hidden Layers***

Though an MLP with just one hidden layer with enough neurons can theoretically model even the most complex functions, for complex problems, deep networks have a much higher efficiency than shallow ones as they can model complex functions using 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 in a hierarchical way. Lower hidden layers in Deep neural networks models the 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).

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 a model is already trained to recognize faces in pictures and a new neural network is needed to be trained to recognize hairstyles, then the model training can be started 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, these can be initialized with 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.

In summary, for many problems the neural network will work just fine with just one or two hidden layers. For more complex problems, number of hidden layers can be ramped up until it starts 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 provided a huge amount of training data is available. But for that purpose it is much more common to reuse parts of a pretrained state-of-the-art network that performs a similar task to make the training a lot faster and will also require much less data.

### ***Number of Neurons per Hidden Layer***

For the number of neurons in the input and output layers, it is determined by the input and output type requirement. For example, the MNIST task requires  $28 \times 28 = 784$  inputs and 10 output neurons.

**Pyramid Approach:** 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 because 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.

**Stretch Pants Approach:** Just like the number of layers, it can also be tried by increasing the number of neurons gradually until the network starts overfitting. Alternatively, a model can also be built with slightly more layers and neurons than actually needed, and then using early stopping and other regularization techniques to prevent it from overfitting too much. This is called the “stretch pants” approach and help avoid bottleneck layers because a layer with too few neurons, will not have enough representational power to preserve all the useful information from the inputs no matter how large the rest of the network is, that information will never be recovered.

### ***Learning Rate, Batch Size, and Other Hyperparameters***

#### **Learning rate**

The learning rate is considered to be the most important hyperparameter and in general, the optimal learning rate is about half of the maximum learning rate. Maximum learning is a learning rate above which the training algorithm diverges. One way to find a good learning rate is to train the model for a few hundred iterations, starting with a very low learning rate (e.g.,  $10^{-5}$ ) and gradually increasing it up to a very large value (e.g., 10). This is done by multiplying the learning rate by a constant factor at each iteration (e.g., by  $(10 / 10^{-5})^{1/500}$  to go from  $10^{-5}$  to 10 in 500 iterations).

If the loss is plotted as a function of the learning rate (using a log scale for the learning rate), its dropping will be seen at first. But after a while, the learning rate will be too large causing loss to shoot back up. The optimal learning rate will be a bit lower than the point at which the loss starts to climb. It is typically about 10 times lower than the turning point. Then model can be reinitialized and trained normally using this good learning rate.

#### **Optimizer**

Choosing a better optimizer than plain old mini-batch gradient descent (and tuning its hyperparameters) is also quite important.

Several advanced optimizers are explained in section [Faster Optimizers](#) in chapter *Training Deep Neural Networks*.



## Batch size

The batch size can have a significant impact on the model's performance and training time. The main benefit of using large batch sizes is that hardware accelerators like GPUs can process them efficiently. Therefore, many researchers and practitioners recommend using the largest batch size that can fit in GPU RAM. But 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. In general, using small batches (of 32, for example) is preferable because small batches led to better models in less training time. Research also showed that it was possible to use very large batch sizes (up to 8,192) along with various techniques such as learning rate warming up (i.e., starting training with a small learning rate, then ramping it up) and to obtain very short training times, without any generalization gap. So, one strategy is to try to using a large batch size, with learning rate warm-up, and if training is unstable or the final performance is disappointing, then try using a small batch size instead.

As optimal learning rate depends the batch size, it should also be updated as well if batch size gets tuned.

## 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 the machine learning task.

## Number of iterations

In most cases, it is recommended to set higher iteration and to use early stopping to stop the training.

## Exercises

[You may refer respective companion source code for answering questions requiring code snippet]

1. Draw an ANN using the original artificial neurons (McCulloch-Pitts artificial neuron) that computes  $A \oplus B$  (where  $\oplus$  represents the XOR operation). Hint:  $A \oplus B = (A \wedge \neg B) \vee (\neg A \wedge B)$ .
2. Why is it generally preferable to use a logistic regression classifier rather than a classic perceptron (i.e., a single layer of threshold logic units trained using the perceptron training algorithm)? How can you tweak a perceptron to make it equivalent to a logistic regression classifier?
3. Why was the sigmoid activation function a key ingredient in training the first MLPs?
4. Write code snippet to build an MLP model to predict house price using California housing dataset and justify the assumptions taken. Consider using Scikit Learn implementation of MLPs.
5. Write code snippet to build an image classifier using Keras Sequential API using Fashion MNIST dataset and to evaluate the model, and also explain rationale behind every assumptions.
6. Name three popular activation functions. Can you draw them?
7. Suppose you have an MLP composed of one input layer with 10 pass-through neurons, followed by one hidden layer with 50 artificial neurons, and finally one output layer with 3 artificial neurons. All artificial neurons use the ReLU activation function.



- a) What is the shape of the input matrix  $\mathbf{X}$ ?
  - b) What are the shapes of the hidden layer's weight matrix  $\mathbf{W}_h$  and bias vector  $\mathbf{b}_h$ ?
  - c) What are the shapes of the output layer's weight matrix  $\mathbf{W}_o$  and bias vector  $\mathbf{b}_o$ ?
  - d) What is the shape of the network's output matrix  $\mathbf{Y}$ ?
  - e) Write the equation that computes the network's output matrix  $\mathbf{Y}$  as a function of  $\mathbf{X}$ ,  $\mathbf{W}_h$ ,  $\mathbf{b}_h$ ,  $\mathbf{W}_o$ , and  $\mathbf{b}_o$ .
8. How many neurons do you need in the output layer if you want to classify email into spam or ham? What activation function should you use in the output layer? If instead you want to tackle MNIST (containing handwritten digits), how many neurons do you need in the output layer, and which activation function should you use? What about the changes required in your network to predict housing prices?
  9. What is backpropagation and how does it work? What is the difference between backpropagation and reverse-mode autodiff?
  10. How can Keras Callback mechanism help handling model overfitting and building better model faster?
  11. Can you list all the hyperparameters you can tweak in a basic MLP? If the MLP overfits the training data, how could you tweak these hyperparameters to try to solve the problem? Write a code snippet to explain hyperparameter tuning using Keras Tuner.

## MODULE 2

### Training Deep Neural Networks

Neural network with just few hidden layers are shallow nets that are generally used for simple problems, but for complex problem, such as detecting hundreds of types of objects in high-resolution images, may need to train a much deeper ANN, perhaps with 10 layers or many more, each containing hundreds of neurons, linked by hundreds of thousands of connections. But increased complexity also bring other concerns some of which are mentioned below.

- **Vanishing/Exploding Gradient:** Lower layers get very hard to train as the gradients ever gets smaller or larger while propagating error gradient from output layer to input layer during backpropagation through DNN.
- **Not-enough Training Data:** Deeper neural nets require relative more training data for better performance.
- **Training Time:** Training time is relatively more.
- **Overfitting:** A larger model (with millions of parameters) may easily get overfitted especially when trained on less data.

### The Vanishing/Exploding Gradient Problems

In second phase during backpropagation, the algorithm computes the gradient of the cost function with regard to each parameter in the network, and uses these gradients to update each parameter with a gradient descent step.

Unfortunately, gradients often get smaller and smaller as the algorithm progresses down to the lower layers. As a result, the gradient descent update leaves the lower layers' connection weights virtually unchanged, and training never converges to a good solution. This is called the *vanishing gradients problem*. In some cases, the opposite can happen: the gradients can grow bigger and bigger until layers get insanely large weight updates and the algorithm diverges. This is the *exploding gradients problem*, which surfaces most often in recurrent neural networks. More generally, deep neural networks suffer from *unstable gradients*; different layers may learn at widely different speeds.

The probable reasons for these problems were found to be the *sigmoid activation function and weight initialization* (normally distributed weights with a mean of 0 and a standard deviation of 1). With these, the variance of outputs of each layer is *much greater* than the variance of its inputs. During forward propagation, the *variance keeps increasing* after each layer until the *activation function saturates* at the top layers. This saturation is actually *made worse* by the fact that the sigmoid function has a *mean of 0.5, not 0* (the hyperbolic tangent function or *tanh* function has a mean of 0 and *behaves slightly better* than the sigmoid function in deep networks).

Below figures shows saturation in sigmoid function as function *saturates at 0 or 1 when input is largely negative or positive, respectively, with derivative close to 0* of curve being flat at both

extremes. Thus, when backpropagation kicks in it has **virtually no gradient to propagate back** through the network, and what little gradient exists keeps **getting diluted** as backpropagation progresses down through the top layers, so there is really **nothing left** for the lower layers.

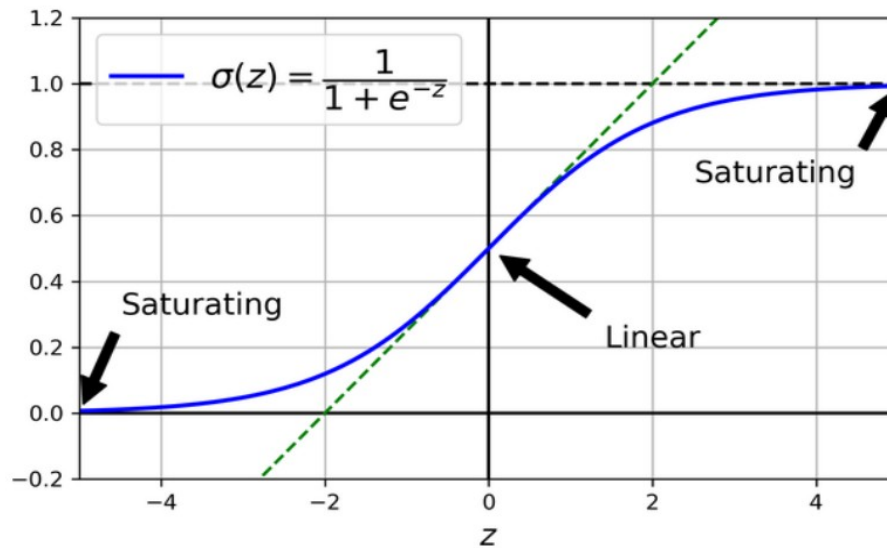


Figure 16: Sigmoid activation function saturation

### Glorot and He Initialization

In one of the proposals to alleviate the above mentioned gradient stability problem, the author Glorot and Bengio suggested for the (1) need for the variance of the outputs of each layer **to be equal** to the variance of its inputs, and for the (2) need for the gradients to have equal variance before and after flowing through a layer in the **reverse direction**. But as it is actually not possible to guarantee both unless the layer has an equal number of inputs and outputs (these numbers are called the **fan-in** and **fan-out** of the layer), authors proposed a **good compromise** that has proven to work very well in practice: the connection weights of each layer must be initialized randomly as described in the equation below where  $fan_{avg} = (fan_{in} + fan_{out}) / 2$ . This initialization strategy is called **Xavier initialization** or **Glorot initialization**.

$$\text{Normal distribution with mean 0 and variance } \sigma^2 = \frac{1}{fan_{avg}}$$

$$\text{Or a uniform distribution between } -r \text{ and } +r \text{ with } r = \sqrt{\frac{3}{fan_{avg}}}$$

Figure 17: Glorot initialization for sigmoid activation function

**LeCun initialization** can be derived if  $fan_{avg}$  is replaced by  $fan_{in}$  in the above equation. Glorot initialization can speed up training considerably, and it is one of the practices that led to the success of deep learning.

By differing only by the scale of the variance and whether they use  $fan_{avg}$  or  $fan_{in}$ , similar strategies for different activation functions were derived and are listed in the below table. For example, the

initialization strategy proposed by Kaiming He for the ReLU activation function and its variants is called **He initialization** or **Kaiming initialization**.

| Initialization | Activation functions                    | $\sigma^2$ (Normal) |
|----------------|---|---------------------|
| Glorot         | None, tanh, sigmoid, softmax            | $1 / fan_{avg}$     |
| He             | ReLU, Leaky ReLU, ELU, GELU, Swish, Mis | $2 / fan_{avg}$     |
| LeCun          | SELU                                    | $1 / fan_{avg}$     |

Table 1: Initialization parameters for each type of activation

By default, Keras uses Glorot initialization with a **uniform distribution**. When a layer is created, initialization can be switched to He initialization by setting `kernel_initializer="he_uniform"` or `kernel_initializer="he_normal"` as shown below.

```
import tensorflow as tf

dense = tf.keras.layers.Dense(50, activation="relu",
                               kernel_initializer="he_normal")
```

He initialization with uniform distribution can be set, for example, with  $fan_{avg}$  using VarianceScaling initializer.

```
he_avg_init = tf.keras.initializers.VarianceScaling(scale=2., mode="fan_avg",
                                                    distribution="uniform")

dense = tf.keras.layers.Dense(50, activation="sigmoid",
                               kernel_initializer=he_avg_init)
```

### Better Activation Functions

Even though sigmoid activation functions roughly works very well in biological neurons, it turns out that **other activation functions behave much better in deep neural networks**—in particular, the **ReLU activation function**, mostly because it **does not saturate** for positive values, and also because it is very **fast to compute**.

Unfortunately, the ReLU activation function is **not perfect**. It suffers from a problem known as the **dying ReLUs**: during training, some neurons effectively “die”, meaning they **stop outputting anything other than 0** for all training instances. In some cases, half of your network’s neurons may get dead, especially for a large learning rate. When this happens, gradient descent does not affect it anymore because the **gradient of the ReLU function is zero when its input is negative**. To solve this problem, a variant of the ReLU function such as the **leaky ReLU** can be used.

### Leaky ReLU

The leaky ReLU activation function is defined as  $LeakyReLU_{\alpha}(z) = \max(\alpha z, z)$  (refer below image). The **hyperparameter  $\alpha$**  defines how much the function “**leaks**”: it is the slope of the function for  $z < 0$  to ensure that **leaky ReLUs never die**. From a comparative study on several variants of the ReLU activation function it was concluded that the **leaky variants always outperforms the strict ReLU**. In fact, setting relatively large leak with  $\alpha = 0.2$  seemed to result better than the smaller leak

with  $\alpha = 0.01$ . Evaluation on **randomized leaky ReLU (RReLU)** where  $\alpha$  was picked randomly in a given range during training and was fixed to an average value during testing, was also done, and it was found that RReLU performed fairly well and seemed to act as a regularizer, reducing the risk of overfitting the training set. Finally, the paper evaluated the **parametric leaky ReLU (PReLU)**, where  $\alpha$  is authorized to be learned during training. PReLU was reported to strongly outperform ReLU on large image datasets, but on smaller datasets it runs the risk of overfitting the training set.

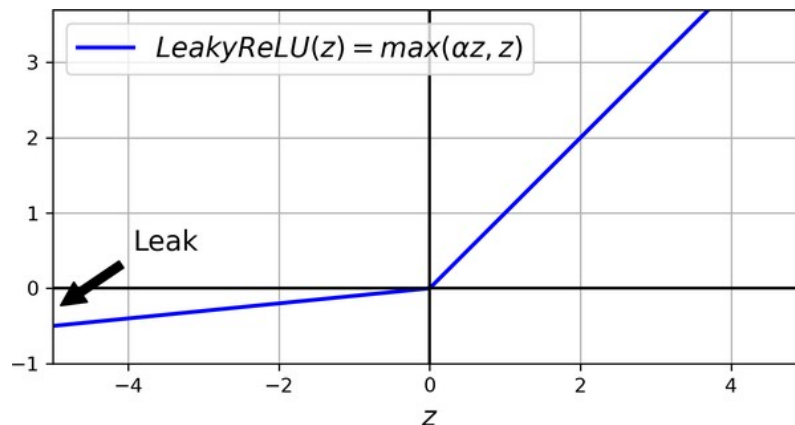


Figure 18: Leaky ReLU: A ReLU with a small slope for negative values

For LeakyReLU and PReLU, He initialization should be used as shown below.

```
leaky_relu = tf.keras.layers.LeakyReLU(alpha=0.2) # defaults to alpha=0.3
dense = tf.keras.layers.Dense(50, activation=leaky_relu,
    kernel_initializer="he_normal")
```

ReLU, leaky ReLU, and PReLU all suffer from the fact that they are **not smooth functions**: their derivatives abruptly change at  $z = 0$ . This sort of discontinuity can make gradient descent bounce around the optimum, and slow down convergence. Some smooth variants of the ReLU activation function such as ELU and SELU address this problem.

## ELU and SELU

Through various experiments it was proved that **exponential linear unit (ELU)** outperformed all the ReLU variants by training time and the neural network performance on the test set.

$$\text{ELU}_{\alpha}(z) = \begin{cases} \alpha (\exp(z) - 1) & \text{if } z < 0 \\ z & \text{if } z \geq 0 \end{cases}$$

Equation 6: ELU activation function

The ELU activation function looks a lot like the ReLU function (refer figure below), with a few major differences:

- It takes on negative values when  $z < 0$ , which allows the unit to have an average output closer to 0 and helps alleviate the vanishing gradients problem. The hyperparameter  $\alpha$  defines the opposite of the value that the ELU function approaches when  $z$  is a large negative number. It is usually set to 1, but can be tweaked like any other hyperparameter.
- It has a nonzero gradient for  $z < 0$ , which avoids the dead neurons problem.
- If  $\alpha$  is equal to 1 then the function is smooth everywhere, including around  $z = 0$ , which helps speed up gradient descent since it does not bounce as much to the left and right of  $z = 0$ .

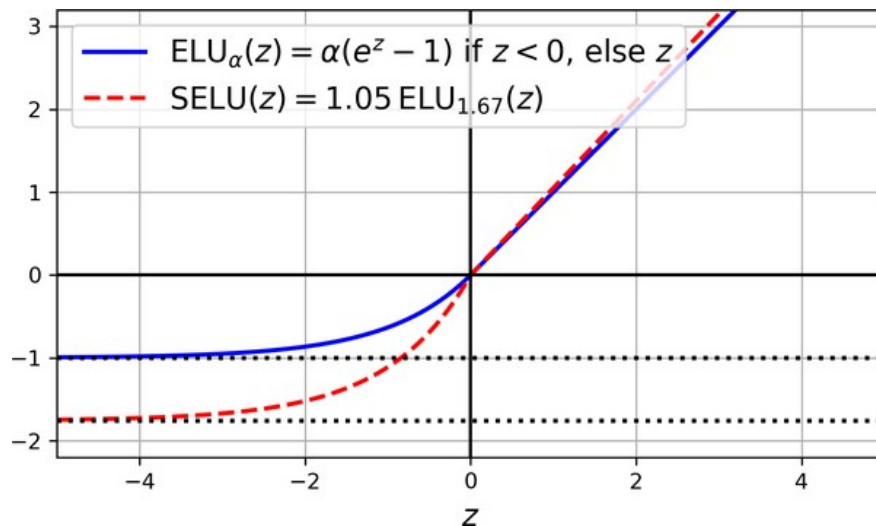


Figure 19: ELU and SELU activation functions

A variant of ELU is **scaled ELU (SELU)** activation function (refer above image) that scales the ELU activation function by about 1.05 times ELU, using  $\alpha \approx 1.67$ . It was shown that a neural network composed exclusively of a stack of MLP dense layers with all hidden layers using SELU activation function (just by setting `activation="selu"` in Keras), then the network will **self-normalize** meaning output of each layer will tend to preserve a mean of 0 and a standard deviation of 1 during training which solves the vanishing/exploding gradients problem. As a result, the **SELU activation function may outperform other activation functions** for MLPs, especially deep ones. There are, however, a few conditions for self-normalization to happen as mentioned below.

- The input features must be standardized: mean 0 and standard deviation 1.
- Every hidden layer's weights must be initialized using LeCun normal initialization. In Keras, this means setting `kernel_initializer="lecun_normal"`.
- The self-normalizing property is only guaranteed with plain MLPs.
- Regularization techniques like  $\ell_1$  or  $\ell_2$  regularization, max-norm, batch-norm, or regular dropout cannot be used.

Because of these significant constraints **SELU did not gain a lot of traction**. Moreover, three more activation functions seem to outperform it quite consistently on most tasks: GELU, Swish, and Mish.

## GELU, Swish, and Mish

### GELU

*GELU* is considered as another smooth variant of the ReLU activation function and is defined as  $GELU(z) = z \Phi(z)$  where  $\Phi$  is the standard Gaussian cumulative distribution function (CDF) and  $\Phi(z)$  corresponds to the probability that a value sampled randomly from a normal distribution of mean 0 and variance 1, is lower than  $z$ . As can be seen in the figure below, *GELU* resembles ReLU as it approaches 0 when its input  $z$  is very negative, and it approaches  $z$  when  $z$  is very positive. It has fairly complex shape by having curvature at every point resulting gradient descent finds it easier to fit complex patterns. However, it is a bit more computationally intensive, and the performance boost it provides is not always sufficient to justify the extra cost. Alternatively, it is approximately equal to  $z\sigma(1.702 z)$  [where  $\sigma$  is the sigmoid function] and this approximation also works very well, and is much faster to compute.

### Swish

*Swish* is another activation function and it outperformed every other function, including *GELU*. It can be generalized by adding an extra hyperparameter  $\beta$  to scale the sigmoid function's input. The generalized Swish function is  $Swish_{\beta}(z) = z\sigma(\beta z)$ , so *GELU* is approximately equal to the generalized Swish function using  $\beta = 1.702$ .  $\beta$  can be used as a hyperparameter as a trainable parameter much like PReLU and let gradient descent optimize it.

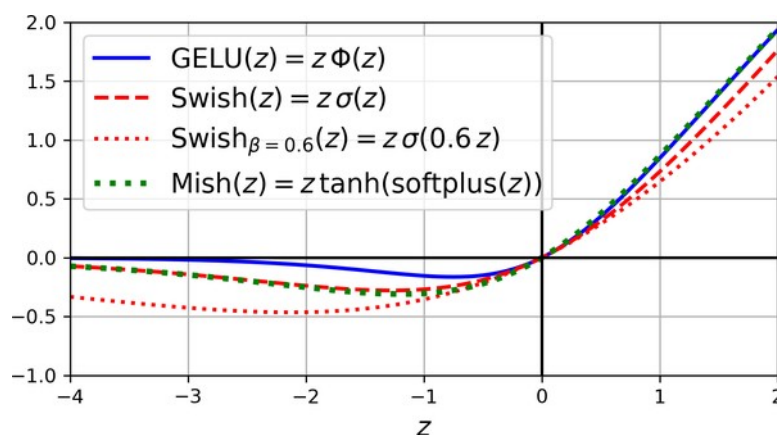


Figure 20: *GELU*, *Swish*, parametrized *Swish*, and *Mish* activation functions

### Mish

Another quite similar activation function is *Mish* and is defined as  $mish(z) = z \tanh(\text{softplus}(z))$ , where  $\text{softplus}(z) = \log(1 + \exp(z))$ . Just like *GELU* and *Swish*, it is a smooth, non-convex, and non-monotonic variant of ReLU and it outperformed other activation functions including *Swish* and *GELU* by a tiny margin. As shown in above figure, *Mish* overlaps almost perfectly with *Swish* when  $z$  is negative, and almost perfectly with *GELU* when  $z$  is positive.

#### TIP for Activation Function Selection

- *ReLU* remains a good default for simple tasks and it is very fast to compute.
- *Swish* is probably a better default for more complex tasks, and even more complex tasks,

parametrized Swish with a learnable  $\beta$  parameter can be tried out.

- Mish may give slightly better results, but it requires a bit more compute.
- To make a trade-off between prediction performance and runtime latency, leaky ReLU, or parametrized leaky ReLU for more complex tasks, can be tried out.

Keras supports GELU and Swish out of the box just by using `activation="gelu"` or `activation="swish"`. However, it does not support Mish or the generalized Swish activation function yet, but can be implemented through custom activation functions.

### Batch Normalization

Although using He initialization along with ReLU or any of its variants can significantly reduce the danger of the vanishing/exploding gradients problems, at the beginning of training, it **doesn't guarantee** that they won't come back during training. **Batch normalization (BN)** is a technique that addresses these problems. The technique consists of adding an operation in the model just **before or after the activation function** of each hidden layer. This operation simply **zero-centers and normalizes each input, then scales and shifts the result** using **two new parameter vectors per layer**: one for scaling, the other for shifting. The model learns the optimal scale and mean of each of the layer's inputs. In many cases, **manual standardization or normalization** of training set would not be required if a BN layer is added as the very first layer of the neural network as layer will look at one batch at a time, and it will rescale and shift each input feature.

In order to zero-center and normalize the inputs, the algorithm needs to estimate each input's mean and standard deviation. It does so by evaluating the mean and standard deviation of the input **over the current mini-batch**. The operations are summarized step by step in the equation below.

$$\begin{aligned} 1 \quad \mu_B &= \frac{1}{m_B} \sum_{i=1}^{m_B} \mathbf{x}^{(i)} \\ 2 \quad \sigma_B^2 &= \frac{1}{m_B} \sum_{i=1}^{m_B} (\mathbf{x}^{(i)} - \mu_B)^2 \\ 3 \quad \mathbf{x}^{(i)} &= \frac{\mathbf{x}^{(i)} - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}} \\ 4 \quad \mathbf{z}^{(i)} &= \gamma \otimes \mathbf{x}^{(i)} + \beta \end{aligned}$$

Equation 7: Batch normalization algorithm

In this algorithm:

- $\mu_B$  is the vector of input means (one mean per input), evaluated over the whole mini-batch B.
- $m_B$  is the number of instances in the mini-batch.



- $\sigma_B$  is the vector of input standard deviations (one standard deviation per input), also evaluated over the whole mini-batch.
- $x^{(i)}$  is the vector of zero-centered and normalized inputs for instance  $i$ .
- $\epsilon$  is a tiny number that avoids division by zero and ensures the gradients don't grow too large (typically  $10^{-5}$ ). This is called a smoothing term.
- $\gamma$  is the output scale parameter vector for the layer (it contains one scale parameter per input).
- $\otimes$  represents element-wise multiplication (each input is multiplied by its corresponding output scale parameter).
- $\beta$  is the output shift (offset) parameter vector for the layer (it contains one offset parameter per input). Each input is offset by its corresponding shift parameter.
- $z^{(i)}$  is the output of the BN operation. It is a rescaled and shifted version of the inputs.

As predictions could be made over a single instance instead of a batch all the time, batch normalization for input during prediction is not feasible. In most implementations of batch normalization, including that of Keras `BatchNormalization` layer, estimate final statistics such as input mean and standard deviation during training by using a moving average of the layer's input means and standard deviations. Four parameter vectors are learned in each batch-normalized layer:  $\gamma$  (the output scale vector) and  $\beta$  (the output offset vector) are learned through regular backpropagation, and  $\mu$  (the final input mean vector) and  $\sigma$  (the final input standard deviation vector) are estimated using an exponential moving average. Note that  $\mu$  and  $\sigma$  are estimated during training, but they are used only after training (to replace the batch input means and standard deviations in the previous equation).

It was proven that batch normalization process not just improves prediction performance, but it also helps reducing vanishing gradient problems, allowing using saturating activation functions such as tanh or even sigmoid activation function, becoming less sensitive to weight initialization, adopting to larger learning rate, acting as a regularizer and making learning process faster.

But there is a runtime penalty as the neural network makes slower predictions due to the extra computations required at each layer. Fortunately, it's often possible to fuse the BN layer with the previous layer after training just by updating the previous layer's weights and biases so that it directly produces outputs of the appropriate scale and offset. For example, if the previous layer computes  $XW + b$ , then the BN layer will compute  $\gamma \otimes (XW + b - \mu) / \sigma + \beta$  (ignoring the smoothing term  $\epsilon$  in the denominator). If we define  $W' = \gamma \otimes W / \sigma$  and  $b' = \gamma \otimes (b - \mu) / \sigma + \beta$ , the equation simplifies to  $XW' + b'$ .

The model with BN may seem to be slower during training, but this could be compensated by the fact that convergence will be much faster with BN meaning it will take fewer epochs to reach the same performance resulting overall shorter wall time for the training.

### Implementing batch normalization with Keras

In the following code snippet, the model with two hidden applies BN after every hidden layer and as the first layer in the model after flattening the input images.

```

model = tf.keras.Sequential([
    tf.keras.layers.Flatten(input_shape=[28, 28]),
    tf.keras.layers.BatchNormalization(),
    tf.keras.layers.Dense(300, activation="relu",
                           kernel_initializer="he_normal"),
    tf.keras.layers.BatchNormalization(),
    tf.keras.layers.Dense(100, activation="relu",
                           kernel_initializer="he_normal"),
    tf.keras.layers.BatchNormalization(),
    tf.keras.layers.Dense(10, activation="softmax")
])

```

The model summary is shown below.

```
model.summary()
```

Model: "sequential"

| Layer (type)                                | Output Shape | Param # |
|---|--------------|---------|
| flatten (Flatten)                           | (None, 784)  | 0       |
| batch_normalization (Batch Normalization)   | (None, 784)  | 3136    |
| dense (Dense)                               | (None, 300)  | 235500  |
| batch_normalization_1 (Batch Normalization) | (None, 300)  | 1200    |
| dense_1 (Dense)                             | (None, 100)  | 30100   |
| batch_normalization_2 (Batch Normalization) | (None, 100)  | 400     |
| dense_2 (Dense)                             | (None, 10)   | 1010    |
| Total params: 271,346                       |              |         |
| Trainable params: 268,978                   |              |         |
| Non-trainable params: 2,368                 |              |         |

Each BN layer in the above example adds four parameters per input:  $\gamma$ ,  $\beta$ ,  $\mu$ , and  $\sigma$  (for example, the first BN layer adds 3,136 parameters, which is  $4 \times 784$ ). Parameters,  $\mu$  and  $\sigma$  are the moving averages and are non-trainable where parameters  $\gamma$  and  $\beta$  are trainable.

### Gradient Clipping

Another technique to mitigate the exploding gradients problem is to clip the gradients during backpropagation so that they never exceed some threshold. This is called *gradient clipping* and is generally used in recurrent neural networks, where using batch normalization is tricky.

In Keras, implementing gradient clipping is just a matter of setting the `clipvalue` or `clipnorm` argument when creating an optimizer as shown below.

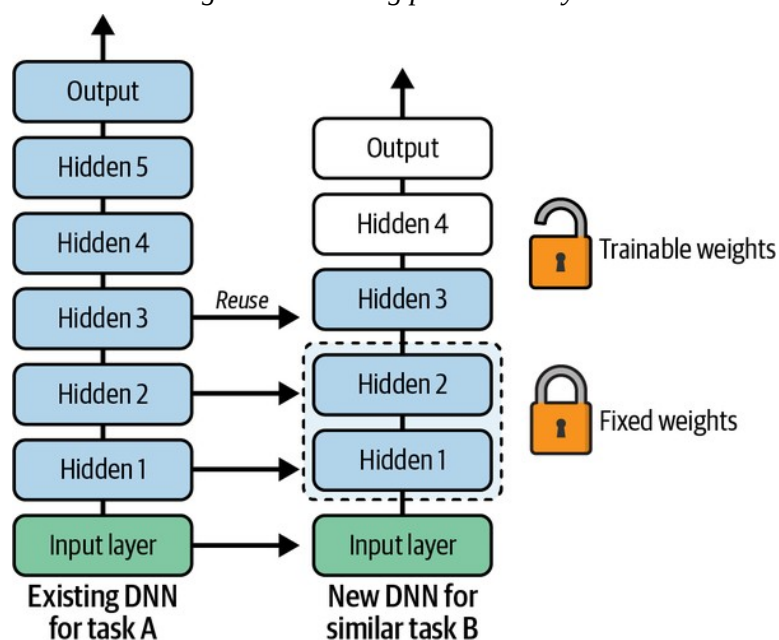
```
optimizer = tf.keras.optimizers.SGD(clipvalue=1.0)
model.compile(..., optimizer=optimizer)
```

This optimizer will clip every component of the gradient vector to a value between `-1.0` and `1.0`. This means that all the partial derivatives of the loss with regard to each and every trainable parameter will be clipped between `-1.0` and `1.0`. The `threshold` is a hyperparameter that can be tuned. Gradient clipping may change the orientation of the gradient vector. For instance, if the original gradient vector is `[0.9, 100.0]`, it points mostly in the direction of the second axis; but once it is clipped by value, it is found to be `[0.9, 1.0]`, which points roughly at the diagonal between the two axes. In practice, this approach works well. If change the direction of the gradient vector by gradient clipping is not expected, clip by norm can be used by setting `clipnorm` instead of `clipvalue`. This will clip the whole gradient if its  $\ell_2$  norm is greater than the set threshold. For example, if `clipnorm=1.0`, then the vector `[0.9, 100.0]` will be clipped to `[0.00899964, 0.9999595]`, preserving its orientation but almost eliminating the first component. If gradients explode during training is observed (by tracking the size of the gradients using TensorBoard), clipping by value or clipping by norm, with different thresholds, can be tried out to find which option performs best on the validation set.

## Reusing Pretrained Layers

It is generally `not a good idea` to train a very large DNN from scratch without first trying to find an existing neural network that accomplishes a similar task at hand. If such a neural network is found, then generally `most of its layers, except for the top ones, can be reused`. This technique is called `transfer learning`. It will not only `speed up training` considerably, but also `requires significantly less training data`.

Figure 21: Reusing pretrained layers



For example, if there is an access to a DNN that was trained to classify pictures into 100 different categories, including animals, plants, vehicles, and everyday objects, and task in hand is now to train a DNN to classify specific types of vehicles, then as these tasks are very similar and even partly overlapping, reusing parts of the existing network should first be tried out. If the input size of the new task is not the same as the ones used in the original task, then usually a **preprocessing step** is added to resize them to the size expected by the original model. More generally, transfer learning will work best when the inputs have **similar low-level features**.

**The output layer** of the original model should usually be replaced because it is most likely not useful at all for the new task, and probably will not have the right number of outputs. Similarly, the **upper hidden layers** of the original model are less likely to be as useful as the lower layers, since the high-level features that are most useful for the new task may differ significantly from the ones that were most useful for the original task. So, right number of layers to reuse needs to be decided.

All the **reused layers should be freezed first** (i.e., make their **weights non-trainable** by the gradient descent), then model should be trained and performance should be observed. Then one or two of the top hidden layers should be tried **unfreezed to make them trainable** and it should be checked if the performance improves. If more training data is available, then performance can be checked by unfreezing more top hidden layers. It is recommended to keep the **learning rate low** when reused layers are unfreezed for not losing the earlier learning stored over fine-tuned weights.

### ***Transfer Learning with Keras***

The following code snippet explains how an existing model A that was created for task A can be reused to create model B to address a similar task B.

First, model A is loaded and a new model is created based on that model's layers. All the layers except for the output layer are reused.

```
[...] # Assuming model A was already trained and saved to "my_model_A"
model_A = tf.keras.models.load_model("my_model_A")

model_B_on_A = tf.keras.Sequential(model_A.layers[:-1])
model_B_on_A.add(tf.keras.layers.Dense(1, activation="sigmoid"))
```

Then the reused layers are freezed during the first few epochs, giving the new layer some time to learn

reasonable weights. To do this, every layer's trainable attribute is set to **False** and the model is compiled.

```
for layer in model_B_on_A.layers[:-1]:
    layer.trainable = False

optimizer = tf.keras.optimizers.SGD(learning_rate=0.001)

model_B_on_A.compile(loss="binary_crossentropy", optimizer=optimizer,
    metrics=["accuracy"])
```

Now the **model is trained for a few epochs**, then the reused layers (which requires compiling the model again) are **unfrozen** and training to fine-tune the reused layers for task B is continued. After unfreezing the reused layers, it is usually a good idea to **reduce the learning rate**, once again to **avoid damaging the reused weights**.

```
history = model_B_on_A.fit(X_train_B, y_train_B, epochs=4, validation_data=(X_val_B, y_val_B))

for layer in model_B_on_A.layers[:-1]:
    layer.trainable = True

optimizer = tf.keras.optimizers.SGD(learning_rate=0.001)

model_B_on_A.compile(loss="binary_crossentropy", optimizer=optimizer, metrics=["accuracy"])
history = model_B_on_A.fit(X_train_B, y_train_B, epochs=16, validation_data=(X_val_B, y_val_B))
```

This model's **test accuracy should be increased** proving transfer learning reduced the error rate by a margin.

```
>>> model_B_on_A.evaluate(X_test_B, y_test_B)
[0.2546142041683197, 0.9384999871253967]
```

Transfer learning does not work very well with small dense networks, presumably because small networks learn few patterns, and dense networks learn very specific patterns, which are unlikely to be useful in other tasks. Transfer learning works best with deep convolutional neural networks, which tend to learn feature detectors that are much more general especially in the lower layers.

### ***Unsupervised Pretraining***

If not much training data is available, pretraining is also possible in unsupervised way by training an unsupervised model such as an autoencoder or a generative adversarial network (GAN) over unlabeled training examples, and then the lower layers of the trained autoencoder or the lower layers of the GAN's discriminator can be reused. Then an output layer can be added on top for the task in hand, and the final network can be fine tuned using supervised learning with the labeled training examples.

### ***Pretraining on an Auxiliary Task***

If much labeled training data is not available, then one last option is to train a first neural network on an auxiliary task for which labeled training data can either be easily obtained or generated, then reuse the lower layers of that network for actual task in hand. The first neural network's lower layers will learn feature detectors that will likely be reusable by the second neural network. For example, if a system needs to be built to recognize faces with only having a few pictures of each individual as training dataset, it is clearly not enough to train a good classifier. However, gathering a lot of pictures of random people on the web is feasible and a neural network can be trained to detect whether or not two different pictures feature the same person. Such a network would learn good feature detectors for faces, so reusing its lower layers would allow to train a good face classifier that uses little training data.

## Faster Optimizers

Training a very large deep neural network can be **painfully slow**. In addition to other ways, a huge speed boost comes from using a **faster optimizer** than the regular gradient descent optimizer. Most popular optimization algorithms are **momentum**, **Nesterov accelerated gradient**, **AdaGrad**, **RMSProp**, and finally **Adam and its variants**. All these optimization techniques rely on the **first-order partial derivatives** (Jacobians).

### Momentum

Regular gradient descent takes small steps when the gradient slope is gentle and big steps when the slope is steep, but it **will never pick up speed**. As a result, regular gradient descent is generally **much slower** to reach the minimum than momentum optimization.

Gradient descent updates the weights  $\theta$  by directly subtracting the gradient of the cost function  $J(\theta)$  with regard to the weights ( $\nabla_{\theta} J(\theta)$ ) multiplied by the learning rate  $\eta$  and the equation is  $\theta \leftarrow \theta - \eta \nabla_{\theta} J(\theta)$ . It **does not care about what the earlier gradients were**. If the local gradient is tiny, it goes very slowly.

Momentum optimization takes a great deal about what previous gradients were: at each iteration, it subtracts the local gradient from the momentum vector  $\mathbf{m}$  (multiplied by the learning rate  $\eta$ ), and it updates the weights by adding this momentum vector (refer equation below).

$$\begin{aligned} 1 \quad & \mathbf{m} \leftarrow \beta \mathbf{m} - \eta \nabla_{\theta} J(\theta) \\ 2 \quad & \theta \leftarrow \theta + \mathbf{m} \end{aligned}$$

*Equation 8: Momentum algorithm*

The gradient is used as an **acceleration, not as a speed**. To simulate some sort of friction mechanism and prevent the momentum from growing too large, the algorithm introduces a **new hyperparameter  $\beta$ , called the momentum**, which must be set between 0 (high friction) and 1 (no friction). A typical momentum value is 0.9.

If the gradient remains constant, the terminal velocity (i.e., the maximum size of the weight updates) is equal to that gradient multiplied by the learning rate  $\eta$  **multiplied by  $1 / (1 - \beta)$**  (ignoring the sign). For example, if  $\beta = 0.9$ , then the terminal velocity is equal to 10 times the gradient times the learning rate, so momentum optimization ends up going 10 times faster than gradient descent! This allows momentum optimization to **escape from plateaus much faster** than gradient descent. When the inputs have very different scales, the cost function will look like an elongated bowl. Gradient descent goes down the steep slope quite fast, but then it takes a very long time to go down the valley. In contrast, momentum optimization will roll down the valley faster and faster until it reaches the bottom (the optimum). In deep neural networks that don't use batch normalization, the upper layers will often end up having inputs with very different scales, so using momentum optimization helps a lot. It can also help roll past local optima.

In Keras, momentum optimization can be implemented as follows.

```
optimizer = tf.keras.optimizers.SGD(learning_rate=0.001, momentum=0.9)
```

### Nesterov Accelerated Gradient

One small variant to momentum optimization, proposed by Yurii Nesterov is almost always **faster than regular momentum optimization**. This *Nesterov accelerated gradient* (NAG) method, also known as *Nesterov momentum optimization*, measures the gradient of the cost function not at the local position  $\theta$  but **slightly ahead in the direction of the momentum**, at  $\theta + \beta \mathbf{m}$  (refer equation below).

$$\begin{array}{l} 1 \quad \mathbf{m} \leftarrow \beta \mathbf{m} - \eta \nabla_{\theta} J(\theta + \beta \mathbf{m}) \\ 2 \quad \theta \leftarrow \theta + \mathbf{m} \end{array}$$

Equation 9: Nesterov accelerated gradient algorithm

This small tweak works because in general the momentum vector will be pointing in the right direction (i.e., toward the optimum), so it will be slightly more accurate to use the gradient measured a bit farther in that direction rather than the gradient at the original position, as shown in below figure where  $\nabla_1$  represents the gradient of the cost function measured at the starting point  $\theta$ , and  $\nabla_2$  represents the gradient at the point located at  $\theta + \beta \mathbf{m}$ .

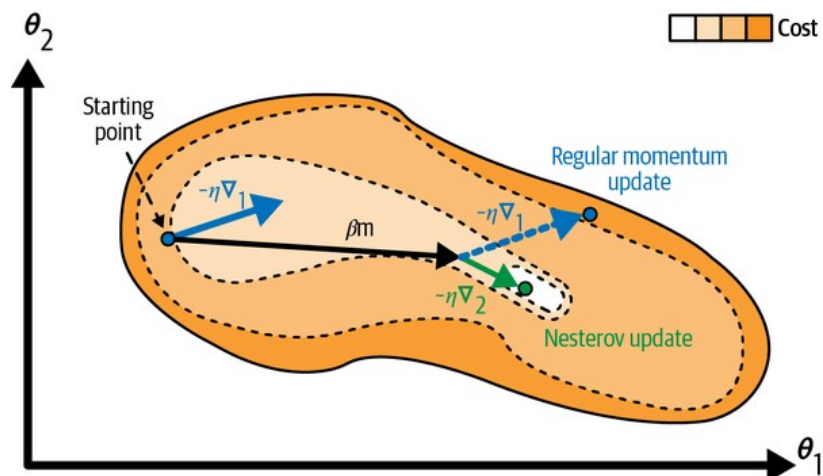


Figure 22: Regular versus Nesterov momentum optimization: the former applies the gradients

As you can see, the Nesterov update **ends up closer to the optimum**. After a while, these small improvements add up and NAG ends up being significantly faster than regular momentum optimization. To use NAG, argument `nesterov` should simply be set to `true` when creating the SGD optimizer:

```
optimizer = tf.keras.optimizers.SGD(learning_rate=0.001, momentum=0.9,
                                     nesterov=True)
```



## AdaGrad

Gradient descent starts by quickly going down the steepest slope and it does not point straight toward the global optimum. It then goes down to the bottom of the valley very slowly. AdaGrad algorithm **corrects its direction earlier to point a bit more toward the global optimum** by scaling down the gradient vector along the steepest dimensions (refer equation below).

$$\begin{aligned} 1 \quad s &\leftarrow s + \nabla_{\theta} J(\theta) \otimes \nabla_{\theta} J(\theta) \\ 2 \quad \theta &\leftarrow \theta - \eta \nabla_{\theta} J(\theta) \oslash \sqrt{s + \varepsilon} \end{aligned}$$

Equation 10: AdaGrad algorithm

The first step accumulates the square of the gradients into the vector  $s$ . In the second step, the gradient vector is scaled down by a factor of  $\sqrt{s + \varepsilon}$  (where  $\varepsilon$  is a smoothing term to avoid division by zero and typically set to  $10^{-10}$ ). This algorithm **decays the learning rate**, but it does so faster for steep dimensions than for dimensions with gentler slopes. This is called an **adaptive learning rate**. It helps point the resulting **updates more directly toward the global optimum** (refer figure below). One additional benefit is that it requires much less tuning of the learning rate hyperparameter  $\eta$ .

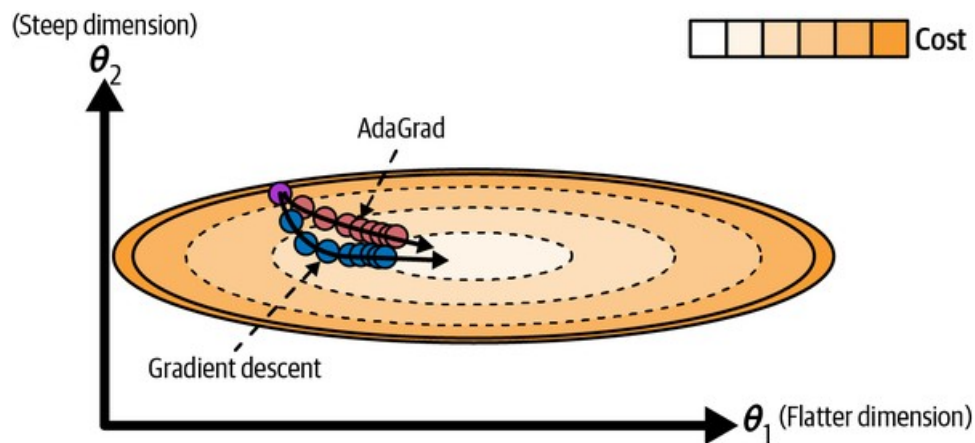


Figure 23: AdaGrad versus gradient descent: the former can correct its direction earlier to point to the optimum

## RMSProp

AdaGrad runs the risk of slowing down a bit too fast and never converging to the global optimum. The RMSProp algorithm fixes this by **accumulating only the gradients from the most recent iterations, as opposed to all the gradients since the beginning of training**. It does so by using **exponential decay** in the first step (refer equation below).

$$\begin{aligned} 1 \quad s &\leftarrow \rho s + (1 - \rho) \nabla_{\theta} J(\theta) \otimes \nabla_{\theta} J(\theta) \\ 2 \quad \theta &\leftarrow \theta - \eta \nabla_{\theta} J(\theta) \oslash \sqrt{s + \varepsilon} \end{aligned}$$

Equation 11: RMSProp algorithm

The **decay rate  $\rho$  (rho)** is typically set to 0.9. Though it is a hyperparameter, but this default value often works well, there might not be any need to tune it at all. This optimizer almost always **performs much better than AdaGrad** and it was the preferred optimization algorithm of many researchers until Adam optimization came around.



## Adam

Adam, which stands for **adaptive moment estimation**, combines the ideas of **momentum optimization** and **RMSProp**: just like momentum optimization, it keeps track of an exponentially decaying average of past gradients; and just like RMSProp, it keeps track of an exponentially decaying average of past squared gradients (refer equation below).

$$1 \quad m \leftarrow \beta_1 m + (1 - \beta_1) \nabla_{\theta} J(\theta)$$

$$2 \quad s \leftarrow \beta_2 s + (1 - \beta_2) \nabla_{\theta} J(\theta) \otimes \nabla_{\theta} J(\theta)$$

$$3 \quad \hat{m} \leftarrow \frac{m}{1 - \beta_1^t}$$

$$4 \quad \hat{s} \leftarrow \frac{s}{1 - \beta_2^t}$$

$$5 \quad \theta \leftarrow \theta + \eta \hat{m} \oslash \sqrt{\hat{s} + \epsilon}$$

Adam's close **similarity to both momentum optimization and RMSProp** can be observed in equation 1, 2 and 5:  $\beta_1$  is **momentum decay hyperparameter** (typically initialized to 0.9) and corresponds to  $\beta$  in momentum optimization, and  $\beta_2$  is **scaling decay hyperparameter** (often initialized to 0.999) and corresponds to  $\rho$  in RMSProp.  $t$  represents the iteration number starting at 1. The only difference is that step 1 computes an exponentially decaying average rather than an exponentially decaying sum, but these are actually equivalent except for a constant factor (the decaying average is just  $1 - \beta_1$  times the decaying sum). Since  $m$  and  $s$ , in steps 3 and 4, are initialized at 0, they will be biased toward 0 at the beginning of training, so these two steps will help boost  $m$  and  $s$  at the beginning of training.

Adam optimization can be used in Keras as follows. Like AdaGrad and RMSProp, it requires less tuning of the learning rate hyperparameter  $\eta$ .

```
optimizer = tf.keras.optimizers.Adam(learning_rate=0.001, beta_1=0.9,
    beta_2=0.999)
```

## AdaMax

In step 2 of Adam equation shown in previous section, Adam accumulates the squares of the gradients in  $s$ . In step 5 (ignoring  $\epsilon$ ), steps 3 and 4, Adam scales down the parameter updates by the square root of  $s$ . In short, Adam scales down the parameter updates by the  $\ell_2$  norm of the time-decayed gradients and replaces the  $\ell_2$  norm with the  $\ell_{\infty}$  norm (max). Specifically, it replaces step 2 with  $s \leftarrow \max(\beta_2 s, |\nabla_{\theta} J(\theta)|)$ , it drops step 4, and in step 5 it scales down the gradient updates by a factor of  $s$ , which is the max of the absolute value of the time-decayed gradients. This can make AdaMax more stable than Adam depending upon dataset, but in general Adam performs better.

```
optimizer = tf.keras.optimizers.Adamax(learning_rate=0.001, beta_1=0.9,
```

```
beta_2=0.999)
```

### Nadam

Nadam optimization is Adam optimization plus the Nesterov trick, so it will often converge slightly faster than Adam. It was found over comparison among many different optimizers on various tasks that Nadam generally outperforms Adam but is sometimes outperformed by RMSProp.

```
optimizer = tf.keras.optimizers.Nadam(learning_rate=0.001,  
momentum=0.9, nesterov=True)
```

### AdamW

AdamW is a variant of Adam that integrates a regularization technique called *weight decay*. Weight decay reduces the size of the model's weights at each training iteration by multiplying them by a decay factor such as 0.99. It is similar to  $\ell_2$  regularization and can be shown mathematically that  $\ell_2$  regularization is equivalent to weight decay when used on SGD. However, when using Adam or its variants,  $\ell_2$  regularization and weight decay are *not* equivalent and in practice, combining Adam with  $\ell_2$  regularization results in models that often don't generalize as well as those produced by SGD. AdamW fixes this issue by properly combining Adam with weight decay.

```
keras.optimizers.AdamW(learning_rate=0.001, weight_decay=0.004, beta_1=0.9,  
beta_2=0.999,
```

Adaptive optimization methods including RMSProp, Adam, AdaMax, Nadam, and AdamW optimization are often great, converging fast to a good solution. However, that they can lead to solutions that generalize poorly on some datasets. As some datasets may not always be a good fit for adaptive optimizers, non-adaptive optimizers such as NAG can be tried out.

| Class                            | Convergence speed | Convergence quality |
|----------------------------------|-------------------|---------------------|
| SGD                              | *                 | ***                 |
| SGD(momentum=...)                | **                | ***                 |
| SGD(momentum=..., nesterov=True) | **                | ***                 |
| Adagrad                          | ***               | * (stops too early) |
| RMSprop                          | ***               | ** or ***           |
| Adam                             | ***               | ** or ***           |
| AdaMax                           | ***               | ** or ***           |
| Nadam                            | ***               | ** or ***           |
| AdamW                            | ***               | ** or ***           |

Table 2: Optimizer comparison (\* is bad, \*\* is average, and \*\*\* is good)

## Avoiding Overfitting Through Regularization

Deep neural networks typically have tens of thousands of parameters, sometimes even millions. This gives them an incredible amount of freedom meaning they can fit a huge variety of complex datasets. But this great flexibility also makes the network prone to overfitting the training set. Regularization is often needed to prevent this.

Early stopping is one of the best regularization techniques and batch normalization, originally designed to solve the unstable gradients problems, also acts like a pretty good regularizer. Other popular regularization techniques for neural networks are  $\ell_1$  and  $\ell_2$  regularization, dropout, and max-norm regularization.

### $\ell_1$ and $\ell_2$ Regularization

$\ell_2$  regularization can be used to constrain a neural network's connection weights where  $\ell_1$  or both  $\ell_1$  and  $\ell_2$  regularization can be used to build a sparse model (with many weights equal to 0). The following code shows that  $\ell_2$  regularization is applied to a Keras layer's connection weights using a regularization factor of 0.01.

```
layer = tf.keras.layers.Dense(100, activation="relu",  
    kernel_initializer="he_normal",  
    kernel_regularizer=tf.keras.regularizers.l2(0.01))
```

The `l2()` function returns a regularizer that will be called at each step during training to compute the regularization loss. It is then added to the final loss. In Keras, both  $\ell_1$  and  $\ell_2$  regularization can be applied by using `tf.keras.regularizers.l1_l2()` by specifying both regularization factors.

$\ell_2$  regularization is fine when using SGD, momentum optimization, and Nesterov momentum optimization, but not with Adam and its variants. If Adam with weight decay needs to be used, then AdamW optimizer should be used considered.

### Dropout

Dropout is one of the most popular regularization techniques for deep neural networks. It is a fairly simple algorithm: at every training step, a random subset of all neurons in one or more layers, except the output layer, has a probability  $p$  of being temporarily “dropped out”, meaning it will be entirely ignored during this training step. These neurons will output 0 at this iteration, but they may be active during the next step (refer figure below). The hyperparameter  $p$  is called the dropout rate, and it is typically set between 10% and 50% for feed-forward neural networks, 20%–30% for recurrent neural networks and closer to 40%–50% for convolutional neural networks. It is to be noted that these drop out gets applied only for training, but not during inference.

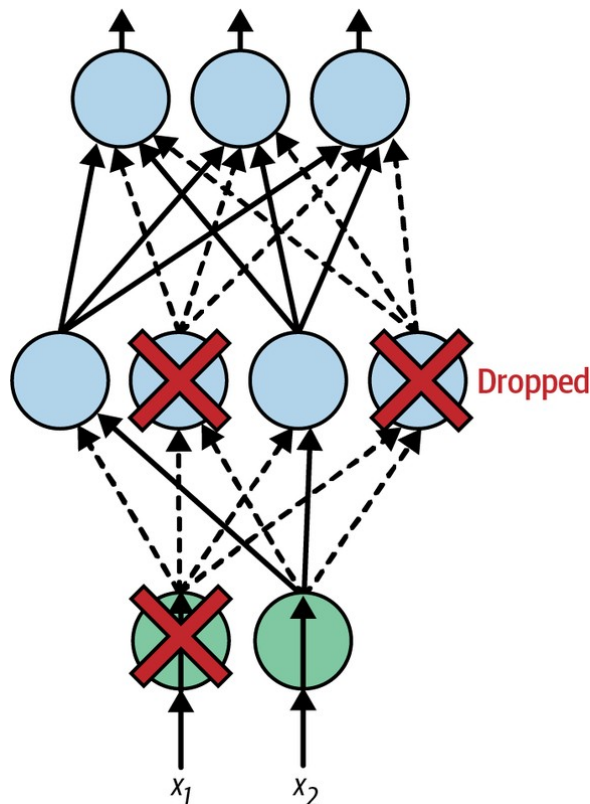


Figure 24: Dropout regularization: at each training iteration, a random subset of all neurons in one or more layers, except the output layer, are dropped out; these neurons output 0 at this iteration (represented by the dashed arrows)

Neurons trained with dropout cannot co-adapt with their neighboring neurons. They get useful as possible on their own by paying attention to each of their input neurons. They end up **being less sensitive** to slight changes in the inputs leading to a more **robust network that generalizes better**.

Another way to understand the power of dropout is to realize that a unique neural network is generated at each training step. If  $N$  is the total number of droppable neurons meaning each neuron can be either present or absent, there could be a total of  $2^N$  possible networks that would be virtually impossible for the same neural network to be sampled twice. These neural networks are obviously not independent because they share many of their weights, but they are nevertheless all different. The resulting neural network can be seen as an averaging ensemble of all these smaller neural networks.

To implement dropout using Keras, layer `tf.keras.layers.Dropout` is used. During training, it **randomly drops some inputs** (setting them to 0) with a mentioned frequency (called *rate* in Keras) at each step and the **remaining inputs are scaled up** by  $1/(1 - \text{rate})$  such that the **sum over all inputs is unchanged**. The following code applies dropout regularization before every dense layer, using a dropout rate of 0.2.

```
model = tf.keras.Sequential([
    tf.keras.layers.Flatten(input_shape=[28, 28]),
    tf.keras.layers.Dropout(rate=0.2),
    tf.keras.layers.Dense(100, activation="relu", kernel_initializer="he_normal"),
    tf.keras.layers.Dropout(rate=0.2),
```

```
tf.keras.layers.Dense(100, activation="relu", kernel_initializer="he_normal"),
tf.keras.layers.Dropout(rate=0.2),
tf.keras.layers.Dense(10, activation="softmax")
])
[...]
```

The dropout rate can be **considered increasing or decreasing** if the model is observed to be overfitting or underfitting the training set, respectively. Dropout does tend to **significantly slow down convergence**, but it often **results in a better model** when tuned properly. To regularize a self-normalizing network based on the SELU activation function, *alpha dropout* should be used as this is a variant of dropout that preserves the mean and standard deviation of its inputs.

### Monte Carlo Dropout

*Monte Carlo (MC) dropout* powerful technique which can boost the performance of any trained dropout model **without having to retrain it or even modify it at all**. It also provides a much **better measure of the model's uncertainty** and improves the reliability of the **model's probability estimates**.

MC dropout can be implemented in just a few lines of code. The following example uses fashion MNIST dataset.

```
import numpy as np

y_probas = np.stack([model(X_test, training=True) for sample in range(100)])
y_proba = y_probas.mean(axis=0)
```

`model(X)` is similar to `model.predict(X)` except it returns a tensor rather than a NumPy array, and setting `training=True` ensures that the Dropout layer remains active. 100 predictions over the test set were made and their average was computed. Each call to the model returns a matrix with one row per instance and one column per class. Because there are 10,000 instances in the test set and 10 classes, this is a matrix of shape [10000, 10]. 100 such matrices were stacked, making `y_probas` a 3D array of shape [100, 10000, 10]. This was averaged over the first dimension (`axis=0`) making `y_proba` an array of shape [10000, 10]. **Averaging over multiple predictions with dropout turned on** gives us a Monte Carlo estimate that is generally more reliable than the **result of a single prediction with dropout turned off**.

The following code predicts the model's confidence on the first test instance and found that to be fairly high (84.4%) indicating this image to belong to class 9 (ankle boot).

```
model.predict(X_test[:1]).round(3)
array([[0., 0., 0., 0., 0., 0.024, 0., 0.132, 0., 0.844]], dtype=float32)
```

This can be compared with the MC dropout prediction as follows.

```
y_proba[0].round(3)
array([0., 0., 0., 0., 0., 0.067, 0., 0.209, 0.001, 0.723], dtype=float32)
```

The model still seems to prefer class 9, but its confidence dropped down to 72.3%, and the estimated probabilities for classes 5 (sandal) and 7 (sneaker) have increased, which makes sense given they're also footwear. Additionally, the standard deviation of the probability estimates can also be looked at as shown below.

```
y_std = y_probab.std(axis=0)
y_std[0].round(3)
array([0. , 0. , 0. , 0.001, 0., 0.096, 0., 0.162, 0.001, 0.183], dtype=float32)
```

Apparently there's **quite a lot of variance in the probability estimates** for class 9: the standard deviation is 0.183, which should be compared to the estimated probability of 0.723. For critical system, moderate prediction probability with moderate to high standard deviation in the predictions should be **treated with caution** because it would not just be an 84.4% confident prediction. Model's MC estimation for accuracy can be measured as follows.

```
y_pred = y_proba.argmax(axis=1)
accuracy = (y_pred == y_test).sum() / len(y_test)

print(accuracy)
0.8717
```

The number of Monte Carlo samples (100 in this example) used here is a **hyperparameter to be tweaked**. The higher it is, the more accurate the predictions and their uncertainty estimates will be, but require more computations increasing latency.

For model containing other layers such as BatchNormalization layers with Dropout layers, these Dropout layers should be replaced with a subclassed implementation of Dropout overriding the `call()` method to force its training argument to True.

```
class MCDropout(tf.keras.layers.Dropout):
    def call(self, inputs, training=False):
        return super().call(inputs, training=True)
```

MC dropout is a great technique that **boosts dropout models** and provides **better uncertainty estimates**. And of course, since it is just regular dropout during training, it also **acts like a regularizer**.

### **Max-Norm Regularization**

Another popular regularization technique for neural networks is called **max-norm regularization**: for each neuron, it constrains the weights  $\mathbf{w}$  of the incoming connections such that  $\|\mathbf{w}\|_2 \leq r$ , where  $r$  is the max-norm hyperparameter and  $\|\cdot\|_2$  is the  $\ell_2$  norm.

Max-norm regularization **does not add a regularization loss term** to the overall loss function. Instead, it is typically implemented by **computing  $\|\mathbf{w}\|_2$**  after each training step and **rescaling  $\mathbf{w}$**  ( $\mathbf{w} \leftarrow \mathbf{w} r / \|\mathbf{w}\|_2$ ), if needed. Reducing  $r$  increases the amount of regularization and helps **reduce overfitting**. Max-norm regularization can also help **alleviate the unstable gradients problems** for model not using batch normalization.

To implement max-norm regularization in Keras, the `kernel_constraint` argument of each hidden layer is set to a `max_norm()` constraint with the appropriate max value as shown below.

```
dense = tf.keras.layers.Dense(
    100, activation="relu", kernel_initializer="he_normal",
    kernel_constraint=tf.keras.constraints.max_norm(1.))
```

After each training iteration, `max_norm()` function gets called with the layer's weights passed onto it and scaled weights are received in return, which then replace the layer's weights. Additionally, the bias terms can also be constrained by setting the `bias_constraint` argument.

The `max_norm()` function has an `axis` argument that defaults to 0 meaning that the max-norm constraint will apply independently to each neuron's weight vector. For max-norm to be used with convolutional layers, the `max_norm()` constraint's `axis` argument should be set appropriately (usually `axis=[0, 1, 2]`).

## Exercises

1. Explain what the problem that Glorot initialization and He initialization aim to fix and how?
2. Is it OK to initialize all the weights to the same value as long as that value is selected randomly using He initialization?
3. Explain how could LeCun initialization be made equivalent to Glorot initialization.
4. Is it OK to initialize the bias terms to 0?
5. Explain how Batch Normalization tries to address vanishing/exploding gradient problems.
6. In which cases would you want to use each of the activation functions we discussed in this chapter?
7. What may happen if you set the momentum hyperparameter too close to 1 (e.g., 0.99999) when using an SGD optimizer?
8. Explain Momentum in context to speeding up training with regular Stochastic Gradient Descent (SDG) optimizer.
9. Why is Nesterov Momentum Optimizer generally faster in convergence than the regular momentum optimizer? Explain the concept with respect to Nesterov Accelerated Gradient (NAG) method.
10. Why is RMSProp optimizer generally faster in convergence than the AdaGrad optimizer.
11. Explain AdamW optimizer and regularization technique it uses to perform generally better than the other regular optimizers.
12. Explain L1 (Lasso) and L2 (Ridge) regularization technique to prevent a deep neural network from being overfitted.

13. Explain the principles behind Dropout regularization technique. Does dropout slow down training? Does it slow down inference? Write code snippet to implement dropout using Keras. Also explain *keep probability*.
14. How does Monte Carlo dropout boosts a dropout model? Write code snippets.
15. Explain Max-Norm regularization technique.



## MODULE 3

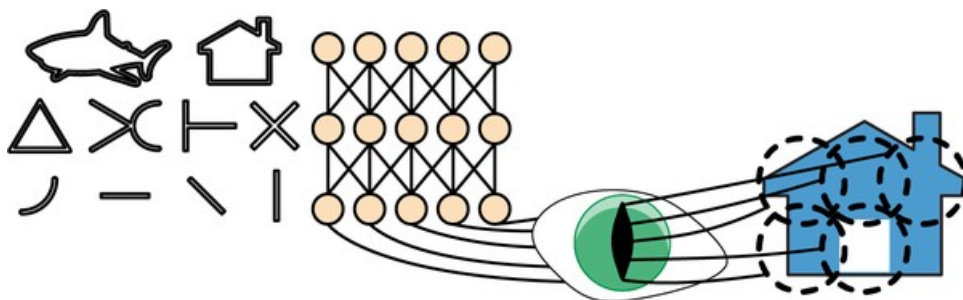
### Convolution Neural Networks

It wasn't until fairly recently that computers were able to reliably perform seemingly trivial tasks such as detecting a puppy in a picture or recognizing spoken words. The reason these tasks **so effortless to us humans** because of the fact that **perception largely takes place outside the realm of our consciousness**, within specialized **visual, auditory, and other sensory modules in our brains**. By the time sensory information reaches our consciousness, it is already adorned with high-level features. **Perception is not trivial at all**, and to understand it, how our sensory modules work should be understood.

**Convolutional neural networks** (CNNs) emerged from the study of the **brain's visual cortex**, and they have been used in computer image recognition since the 1980s. Over the last 10 years, thanks to the increase in computational power, the amount of available training data, and the tricks for training deep nets, CNNs have managed to achieve **superhuman performance on some complex visual tasks**. They power image search services, self-driving cars, automatic video classification systems, and more. Moreover, CNNs are not restricted to visual perception: they are also successful at many other tasks, such as voice recognition and natural language processing.

#### The Architecture of the Visual Cortex

A series of experiments were performed on cats in 1958 and 1959 (and a few years later on monkeys) revealed crucial insights into the structure of the visual cortex. The experiments showed that many neurons in the visual cortex have a **small local receptive field**, meaning they **react only to visual stimuli located in a limited region of the visual field** (see figure, in which the local receptive fields of five neurons are represented by dashed circles). **The receptive fields of different neurons may overlap**, and together they tile the whole visual field.



*Figure 25: Biological neurons in the visual cortex respond to specific patterns in small regions of the visual field called receptive fields; as the visual signal makes its way through consecutive brain modules, neurons respond to more complex patterns in larger receptive fields*

Moreover, it was also shown that some neurons react only to images of horizontal lines, while others react only to lines with different orientations (two neurons may have the same receptive field but react to different line orientations). It was also noticed that **some neurons have larger receptive fields**, and they **react to more complex patterns** that are combinations of the lower-level patterns. These observations led to the idea that the higher-level neurons are based on the outputs of neighboring lower-level neurons (in the above figure, it can be noticed that each neuron is

connected only to nearby neurons from the previous layer). This powerful architecture is able to detect all sorts of complex patterns in any area of the visual field.

These studies of the visual cortex inspired the *neocognitron*, introduced in 1980, which gradually evolved into what we now call convolutional neural networks. An important milestone was a 1998 paper by Yann LeCun et al. that introduced the famous **LeNet-5** architecture, which became widely used by banks to recognize handwritten digits on checks. This architecture has some building blocks such as fully connected layers and sigmoid activation functions, but it also introduces two new building blocks: **convolutional layers** and **pooling layers**.

**NOTE:** The reason that a deep neural network with fully connected layers cannot just be used for large image recognition tasks is that though it works fine for small images (e.g., MNIST), it breaks down for larger images because of the huge number of parameters it requires. For example, a  $100 \times 100$  pixel image has 10,000 pixels, and if the first layer has just 1,000 neurons (which already severely restricts the amount of information transmitted to the next layer), this means a total of 10 million connections. And that's just the first layer. CNNs solve this problem using partially connected layers and weight sharing.

## Convolution Layers

The **most important building block** of a CNN is the **convolutional layer**: neurons in the first convolutional layer are **not connected** to every single pixel in the input image, but only to pixels in their receptive fields (see figure below). In turn, each neuron in the second convolutional layer is **connected only to neurons located within a small rectangle in the first layer**. This architecture allows the network to concentrate on **small low-level features** in the first hidden layer, then assemble them into **larger higher-level features** in the next hidden layer, and so on. This hierarchical structure is common in real-world images, which is one of the reasons why CNNs work so well for image recognition. In a CNN, each layer is represented in 2D, which makes it easier to match neurons with their corresponding inputs. Unlike full connected neural network, the input image does not need to be flattened to 1D before feeding to the neural network.

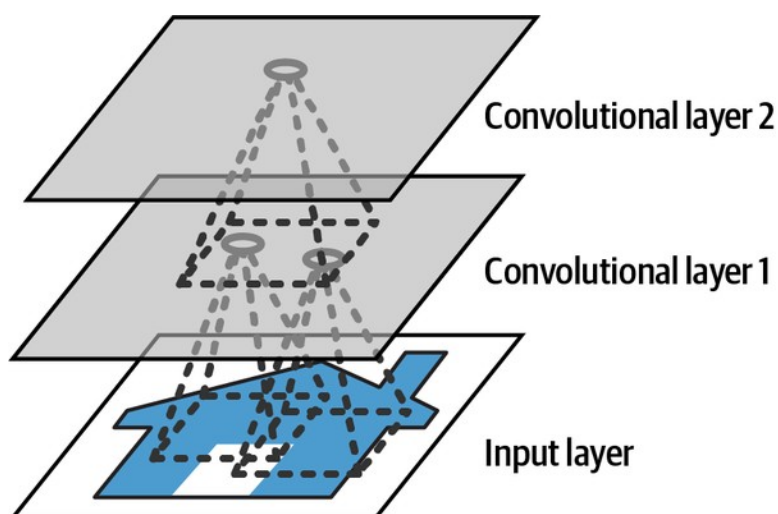


Figure 26: CNN layers with rectangular local receptive fields

A neuron located in row  $i$ , column  $j$  of a given layer is connected to the outputs of the neurons in the previous layer located in rows  $i$  to  $i + f_h - 1$ , columns  $j$  to  $j + f_w - 1$ , where  $f_h$  and  $f_w$  are the height and width of the receptive field or kernel (see below figure). In order for a layer to have the same height and width as the previous layer, it is common to add zeros around the inputs, as shown in the below diagram. This is called **zero padding**.

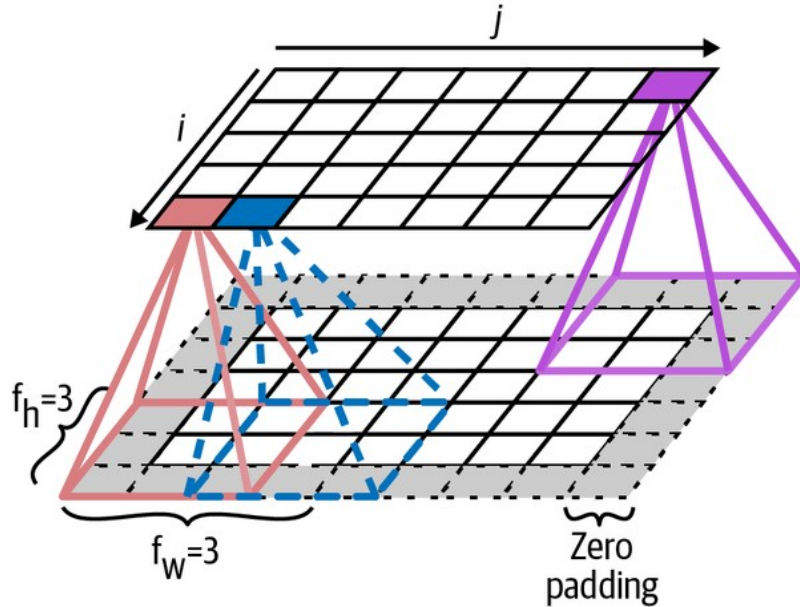


Figure 27: Connections between layers and zero padding

It is also possible to connect a large input layer to a much smaller layer by spacing out the receptive fields, as shown in the below figure. This **dramatically reduces the model's computational complexity**. The **horizontal or vertical step size** from one receptive field to the next is called the **stride**. In the below diagram, a  $5 \times 7$  input layer (plus zero padding) is connected to a  $3 \times 4$  layer, using  $3 \times 3$  receptive fields and a stride of 2 (in this example the stride is the same in both directions, but it does not have to be so). A neuron located in row  $i$ , column  $j$  in the upper layer is connected to the outputs of the neurons in the previous layer located in rows  $i \times s_h$  to  $i \times s_h + f_h - 1$ , columns  $j \times s_w$  to  $j \times s_w + f_w - 1$ , where  $s_h$  and  $s_w$  are the vertical and horizontal strides.

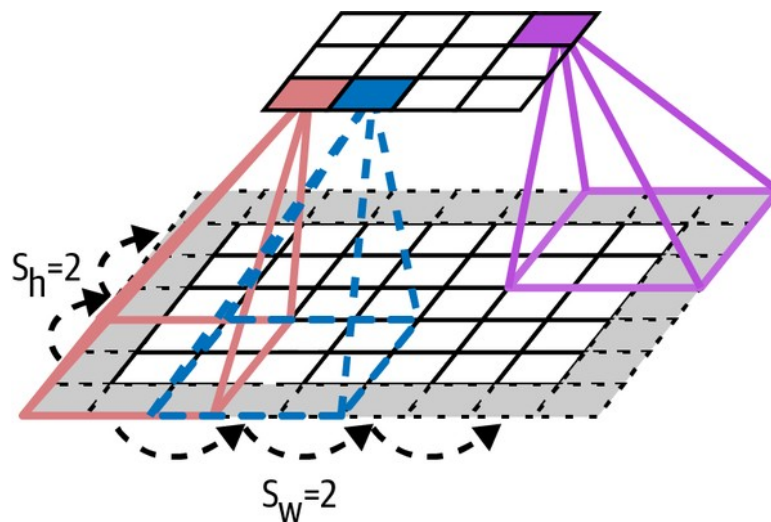


Figure 28: Reducing dimensionality using a stride of 2

### Filters

A neuron's weights can be represented as a small image the size of the receptive field. Possible sets of weights are called **filters** (or convolution kernels, or just **kernels**) represented generally as square ( $n \times n$ ) matrix (where  $n$  is length of one side of the matrix) of weights to be combined linearly with the input and a bias term to be added. A convolution layer full of neurons using the same filter outputs a **feature map**, which highlights the areas in an input image that activate the filter the most. But the filters do not need to be defined manually, instead, during training the convolutional layer will automatically learn the most useful filters for its task, and the layers above will learn to combine them into more complex patterns.

### Stacking Multiple Feature Maps

A convolutional layer generally contains multiple filters and outputs one feature map per filter (see below figure). More accurately filters are represented in 3D – one additional dimension for multiple filters each of which is represented in 2D. It has one neuron per pixel in each feature map, and all neurons within a given feature map share the same parameters (i.e., the same kernel and bias term) resulting dramatic reduction of the number of parameters in the model. Neurons in different feature maps use different parameters. A convolutional layer simultaneously applies multiple trainable filters to its inputs, making it capable of detecting multiple features anywhere in its inputs meaning once the CNN has learned to recognize a pattern in one location, it can recognize it in any other location. In contrast, once a fully connected neural network has learned to recognize a pattern in one location, it can only recognize it in that particular location.

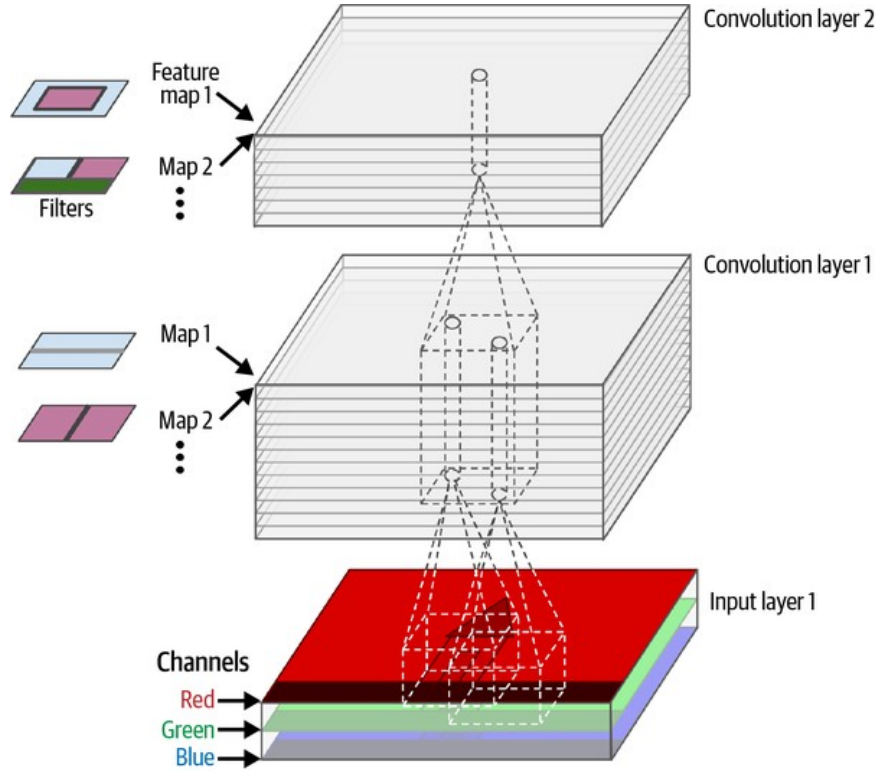


Figure 29: Two convolutional layers with multiple filters each (kernels), processing a color image with three color channels; each convolutional layer outputs one feature map per filter

As mentioned earlier, a neuron located in row  $i$ , column  $j$  of the feature map  $k$  in a given convolutional layer  $l$  is connected to the outputs of the neurons in the previous layer  $l-1$ , located in rows  $i \times s_h$  to  $i \times s_h + f_h - 1$  and columns  $j \times s_w$  to  $j \times s_w + f_w - 1$ , across all feature maps (in layer  $l-1$ ). Note that, within a layer, all neurons located in the same row  $i$  and column  $j$  but in different feature maps are connected to the outputs of the exact same neurons in the previous layer.

Below equation summarizes the preceding explanations in mathematical equation: it shows how to compute the output of a given neuron in a convolutional layer. It calculates the weighted sum of all the inputs, plus the bias term.

$$z_{i,j,k} = b_k + \sum_{u=0}^{f_h-1} \sum_{v=0}^{f_w-1} \sum_{k'=0}^{f_{n'}-1} x_{i',j',k'} \times w_{u,v,k',k} \text{ with } i' = i \times s_h + u \text{ and } j' = j \times s_w + v$$

Equation 12: Computing the output of a neuron in a convolutional layer

In this equation:

- $z_{i,j,k}$  is the output of the neuron located in row  $i$ , column  $j$  in feature map  $k$  of the convolutional layer (layer  $l$ ).
- $s_h$  and  $s_w$  are the vertical and horizontal strides,  $f_h$  and  $f_w$  are the height and width of the receptive field, and  $f_{n'}$  is the number of feature maps in the previous layer (layer  $l-1$ ).

- $x_{i',j',k'}$  is the output of the neuron located in layer  $l - 1$ , row  $i'$ , column  $j'$ , feature map  $k'$  (or channel  $k'$  if the previous layer is the input layer).
- $b_k$  is the bias term for feature map  $k$  (in layer  $l$ ). This can be considered as a knob that tweaks the overall brightness of the feature map  $k$ .
- $w_{u,v,k',k}$  is the connection weight between any neuron in feature map  $k$  of the layer  $l$  and its input located at row  $u$ , column  $v$  (relative to the neuron's receptive field), and feature map  $k'$ .

### Implementing Convolution Layers with Keras

Following code-snippet preprocesses the input image by center cropping and rescaling.

```
from sklearn.datasets import load_sample_images
import tensorflow as tf

// Preprocesses a couple of sample images
images = load_sample_images()["images"]
images = tf.keras.layers.CenterCrop(height=70, width=120)(images)
images = tf.keras.layers.Rescaling(scale=1 / 255)(images)

images.shape
TensorShape([2, 70, 120, 3])
```

First dimension indicates the number of input images – two in this case. Second and third indicate cropped image size and the third dimension indicates the number of channels in the input – three in this case for (RGB) colour image input.

### Memory Requirements

Another challenge with CNNs is that the convolutional layers **require a huge amount of RAM**. This is especially true during training, because the reverse pass of backpropagation requires all the intermediate values computed during the forward pass. For example, consider a convolutional layer with  $200 \ 5 \times 5$  filters, with stride 1 and "same" padding. If the input is a  $150 \times 100$  RGB image (three channels), then the **number of parameters** is  $(5 \times 5 \times 3 + 1) \times 200 = 15,200$  (the + 1 corresponds to the bias terms), which is fairly small compared to a fully connected layer. However, each of the 200 feature maps contains  $150 \times 100$  neurons, and each of these neurons needs to compute a weighted sum of its  $5 \times 5 \times 3 = 75$  inputs: that's a total of 225 million ( $75 \times 150 \times 100 \times 200$ ) **float multiplications**. Not as bad as a fully connected layer, but still quite computationally intensive.

Moreover, if the feature maps are represented using **32-bit floats**, then the convolutional layer's output will occupy  $200 \times 150 \times 100 \times 32 = 96$  million bits (12 MB) of **RAM for one instance**. If a **training batch** contains 100 instances, then this layer will use up 1.2 GB of RAM. During inference (i.e., when making a prediction for a new instance) the RAM occupied by one layer can be released as soon as the next layer has been computed, so you only need as much RAM as required by two consecutive layers. But **during training everything computed during the forward pass needs to be**



preserved for the reverse pass, so the amount of RAM needed is (at least) the total amount of RAM required by all layers.

## Pooling Layers

The goal of a pooling layer is to **subsample** (i.e., shrink) the input image in order to **reduce** the computational load, the memory usage, and the number of parameters (thereby limiting the risk of overfitting). Just like in convolutional layers, each neuron in a pooling layer is connected to the outputs of a limited number of neurons in the previous layer, located within a small rectangular receptive field. Just like convolution layer, **size, the stride, and the padding type** of a pooling layer need to be defined. However, a pooling neuron has **no weights**; all it does is aggregate the inputs using an **aggregation function such as the max or mean**. Below figure shows a **max pooling layer**, which is the most common type of pooling layer. In this example, we use a  $2 \times 2$  **pooling kernel**, with a stride of 2 and no padding. Only the **max input value** in each receptive field makes it to the next layer, while the other inputs are dropped. For example, in the lower-left receptive field in the figure, the input values are 1, 5, 3, 2, so only the max value, 5, is propagated to the next layer. Because of the stride of 2, the output image has **half the height and half the width** of the input image (rounded down since no padding is used). A pooling layer typically works on every input channel independently, so the output depth (i.e., the number of channels) is the same as the input depth.

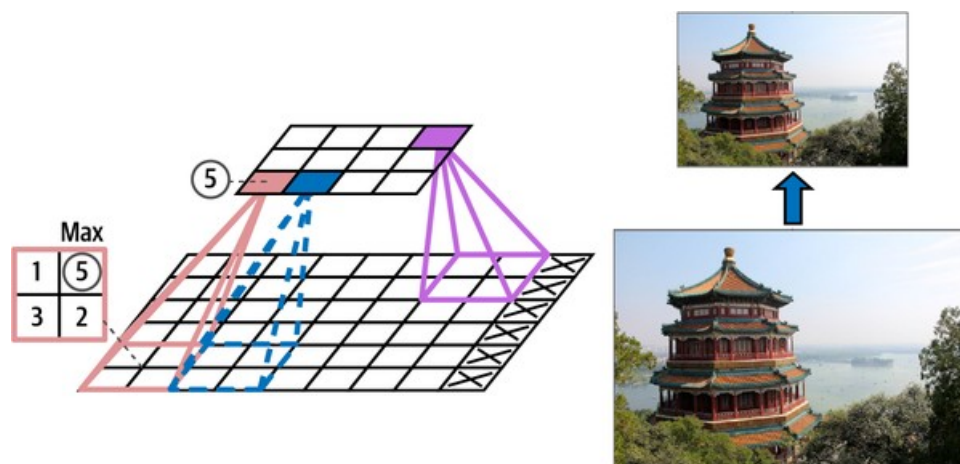


Figure 30: Max pooling layer ( $2 \times 2$  pooling kernel, stride 2, no padding)

Other than **reducing computations, memory usage, and the number of parameters**, a max pooling layer also introduces some level of *invariance* to small translations, as shown in below figure. The assumptions here are the bright pixels have a lower value than dark pixels, and three images (A, B, C) are going through a max pooling layer with a  $2 \times 2$  kernel and stride 2. Images B and C are the same as image A, but shifted by one and two pixels to the right. As seen, the outputs of the max pooling layer for images A and B are identical. This is what **translation invariance** means. For image C, the output is different: it is shifted one pixel to the right (but there is still 50% invariance). Moreover, max pooling offers a small amount of **rotational invariance** and a slight **scale invariance**. Such invariance (even if it is limited) can be useful in cases where the prediction should not depend on these details, such as in classification tasks.

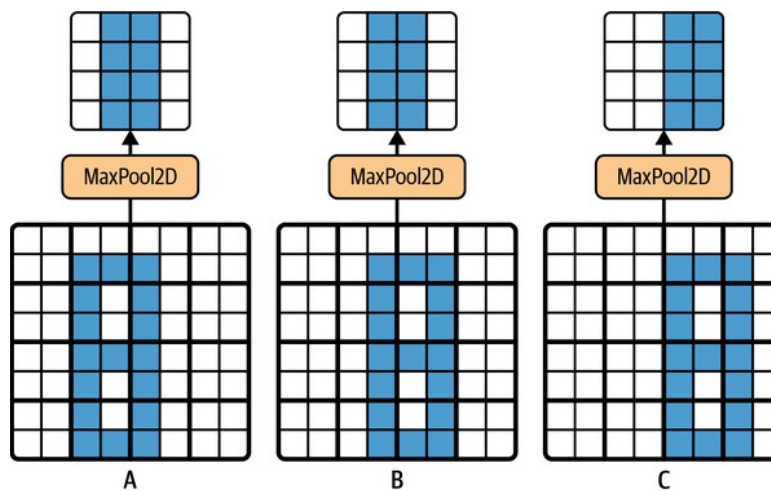


Figure 31: Invariance to small translations

However, max pooling has some **downsides** too. It's obviously very **destructive**: even with a tiny  $2 \times 2$  kernel and a stride of 2, the output will be **two times smaller** in both directions (so its area will be four times smaller), simply **dropping 75%** of the input values. And in some applications, **invariance is not desirable**. Take semantic segmentation (the task of classifying each pixel in an image according to the object that pixel belongs to, which we'll explore later in this chapter): obviously, if the input image is translated by one pixel to the right, the output should also be translated by one pixel to the right. The goal in this case is **equivariance**, not invariance: a small change to the inputs should lead to a corresponding small change in the output.

## Implementing Pooling Layers with Keras

The following code creates a `MaxPooling2D` layer, alias `MaxPool2D`, using a  $2 \times 2$  kernel. The strides default to the kernel size, so this layer uses a stride of 2 (horizontally and vertically). By default, it uses "valid" padding (i.e., no padding at all).

```
max_pool = tf.keras.layers.MaxPool2D(pool_size=2)
```

To create an average pooling layer, just use `AveragePooling2D`, alias `AvgPool2D` needs to be used, instead of `MaxPooling2D`. As you might expect, it works exactly like a max pooling layer, except it computes the mean rather than the max. As compared to Average pooling layers, **people mostly use max pooling layers now**, as they generally **perform better** because max pooling **preserves only the strongest features**, getting rid of all the meaningless ones, so the next layers get a cleaner signal to work with. Moreover, max pooling offers stronger **translation invariance** than average pooling, and it requires slightly less compute.

**Depthwise Pooling:** Max pooling and average pooling can also be performed along the **depth dimension instead of the spatial dimensions**, although it's not as common. This can allow the CNN to **learn to be invariant to various features**. For example, it could learn multiple filters, each detecting a **different rotation of the same pattern** (such as handwritten digits; see below figure), and the depthwise max pooling layer would ensure that the output is the same regardless of the rotation.



The CNN could similarly learn to be **invariant to anything: thickness, brightness, skew, color, and so on**. Keras does not include a depthwise max pooling layer, but it can be included by implementing a custom layer for that.

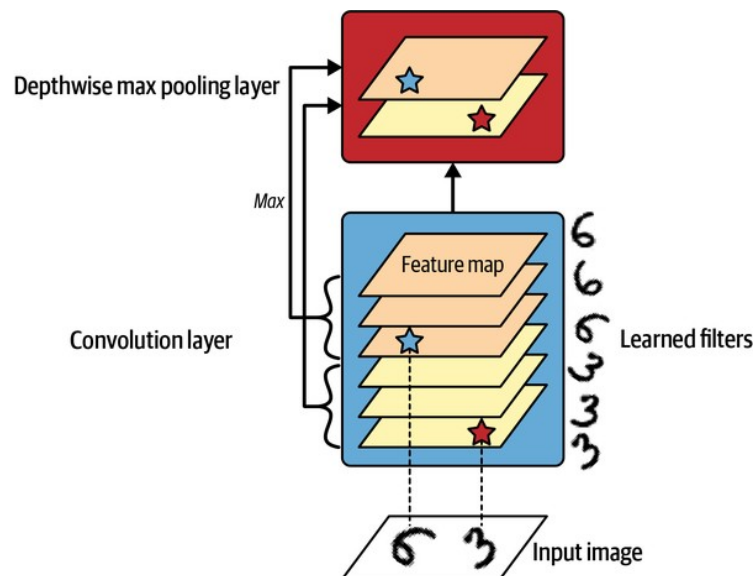


Figure 32: Depthwise max pooling can help the CNN learn to be invariant (to rotation in this case)

One last type of pooling layer that is often seen in modern architectures is the **global average pooling layer**. It works very differently: all it does is **compute the mean of each entire feature map** (it's like an average pooling layer using a pooling kernel with the same spatial dimensions as the inputs). This means that it just **outputs a single number per feature map and per instance**. Although this is of course **extremely destructive** (most of the information in the feature map is lost), it can be useful **just before the output layer**. In Keras, this layer can be created by using the `GlobalAveragePooling2D` class, alias `GlobalAvgPool2D`.

```
global_avg_pool = tf.keras.layers.GlobalAvgPool2D()
```

## CNN Architectures

Typical CNN architectures stack a few convolutional layers (each one generally followed by a ReLU layer), then a pooling layer, then another few convolutional layers (+ReLU), then another pooling layer, and so on. The image gets smaller and smaller as it progresses through the network, but it also typically gets **deeper and deeper** (i.e., with more feature maps), thanks to the convolutional layers (see below figure). At the top of the stack, a regular feedforward neural network is added, composed of a few fully connected layers (+ReLU), and the final layer outputs the prediction (e.g., a softmax layer that outputs estimated class probabilities).

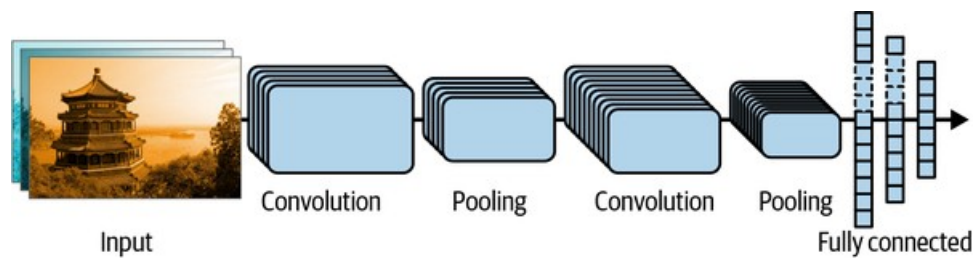


Figure 33: Typical CNN architecture

Here is how a basic CNN can be implemented to tackle the Fashion MNIST dataset:

```
from functools import partial

DefaultConv2D = partial(tf.keras.layers.Conv2D, kernel_size=3, padding="same",
                        activation="relu", kernel_initializer="he_normal")

model = tf.keras.Sequential([
    DefaultConv2D(filters=64, kernel_size=7, input_shape=[28, 28, 1]),
    tf.keras.layers.MaxPool2D(),
    DefaultConv2D(filters=128),
    DefaultConv2D(filters=128),
    tf.keras.layers.MaxPool2D(),
    DefaultConv2D(filters=256),
    DefaultConv2D(filters=256),
    tf.keras.layers.MaxPool2D(),
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(units=128, activation="relu", kernel_initializer="he_normal"),
    tf.keras.layers.Dropout(0.5),
    tf.keras.layers.Dense(units=64, activation="relu", kernel_initializer="he_normal"),
    tf.keras.layers.Dropout(0.5),
    tf.keras.layers.Dense(units=10, activation="softmax")
])
```

Discussion on the above code snippet:

- First layer is a `DefaultConv2D` with 64 fairly large filters ( $7 \times 7$ ) with the default stride of 1 because the input images are not very large. It also sets `input_shape=[28, 28, 1]`, because the images are `single color channel` (i.e., grayscale)  $28 \times 28$  pixels.
- Max pooling layer that uses the `default pool size of 2`, so it divides each spatial dimension by a factor of 2.
- Same structure is `repeated twice`: two convolutional layers followed by a max pooling layer. For larger images, this structure could be repeated several more times. The number of repetitions is a hyperparameter to tune.
- The number of `filters doubles` while climbing up the CNN toward the output layer (it is initially 64, then 128, then 256) because the number of low-level features is often fairly low (e.g., small circles, horizontal lines), but there are many `different ways to combine` them into higher-level features. It is a common practice to `double the number of filters` after each pooling layer without fear of exploding the number of parameters, memory usage, or computational load, since a pooling layer divides each spatial dimension by a factor of 2.

- Next is the **fully connected network**, composed of two hidden dense layers and a dense output layer of 10 units with softmax activation since it's a classification task with 10 classes. Inputs are then flatten just before the first dense layer, since it expects a 1D array of features for each instance. Two **dropout layers** are also added with 50% dropout rate for each, to reduce overfitting.

Over the years, variants of this fundamental architecture have been developed, leading to amazing advances in the field. A good measure of this progress is the error rate in competitions such as the **ILSVRC ImageNet challenge**. The images are fairly large (e.g., 256 pixels high) and there are 1,000 classes.

First the classical LeNet-5 architecture (1998) will be looked at, then several winners of the ILSVRC challenge: AlexNet (2012), GoogLeNet (2014), ResNet (2015), and SENet (2017). Along the way, few more architectures, including Xception, ResNeXt, DenseNet, MobileNet, CSPNet, and EfficientNet will also be discusse.

### LeNet-5

The LeNet-5 architecture is perhaps the most widely known CNN architecture created by Yann LeCun in 1998 and has been widely used for handwritten digit recognition (MNIST). It is composed of the layers shown in the table below. It has stack of convolutional layers and pooling layers, followed by a dense network. Perhaps the main difference with more modern classification CNNs is the activation functions: today, **ReLU instead of tanh** and **softmax instead of RBF** are used.

| Layer | Type            | Maps | Size    | Kernel size | Stride | Activation |
|-------|-----------------|------|---------|-------------|--------|------------|
| Out   | Fully connected | —    | 10      | —           | —      | RBF        |
| F6    | Fully connected | —    | 84      | —           | —      | tanh       |
| C5    | Convolution     | 120  | 1 × 1   | 5 × 5       | 1      | tanh       |
| S4    | Avg pooling     | 16   | 5 × 5   | 2 × 2       | 2      | tanh       |
| C3    | Convolution     | 16   | 10 × 10 | 5 × 5       | 1      | tanh       |
| S2    | Avg pooling     | 6    | 14 × 14 | 2 × 2       | 2      | tanh       |
| C1    | Convolution     | 6    | 28 × 28 | 5 × 5       | 1      | tanh       |
| In    | Input           | 1    | 32 × 32 | —           | —      | —          |

Figure 34: LeNet-5 architecture

### AlexNet

The AlexNet CNN architecture won the 2012 ILSVRC challenge by a large margin. It is similar to LeNet-5, only **much larger and deeper**, and it was the first to **stack convolutional layers directly on top of one another**, instead of stacking a pooling layer on top of each convolutional layer. Table below presents this architecture. To reduce overfitting, two regularization techniques were used. First, they applied **50% dropout** during training to the outputs of layers F9 and F10. AlexNet also uses a competitive normalization step immediately after the ReLU step of layers C1 and C3, called **local response normalization (LRN)**: the most strongly activated neurons inhibit other neurons located at the same position in neighboring feature maps. Such competitive activation has been

observed in biological neurons. This encourages different feature maps to specialize, pushing them apart and forcing them to explore a wider range of features, ultimately improving generalization.

| Layer | Type            | Maps    | Size             | Kernel size    | Stride | Padding | Activation |
|-------|-----------------|---------|------------------|----------------|--------|---------|------------|
| Out   | Fully connected | —       | 1,000            | —              | —      | —       | Softmax    |
| F10   | Fully connected | —       | 4,096            | —              | —      | —       | ReLU       |
| F9    | Fully connected | —       | 4,096            | —              | —      | —       | ReLU       |
| S8    | Max pooling     | 256     | $6 \times 6$     | $3 \times 3$   | 2      | valid   | —          |
| C7    | Convolution     | 256     | $13 \times 13$   | $3 \times 3$   | 1      | same    | ReLU       |
| C6    | Convolution     | 384     | $13 \times 13$   | $3 \times 3$   | 1      | same    | ReLU       |
| C5    | Convolution     | 384     | $13 \times 13$   | $3 \times 3$   | 1      | same    | ReLU       |
| S4    | Max pooling     | 256     | $13 \times 13$   | $3 \times 3$   | 2      | valid   | —          |
| C3    | Convolution     | 256     | $27 \times 27$   | $5 \times 5$   | 1      | same    | ReLU       |
| S2    | Max pooling     | 96      | $27 \times 27$   | $3 \times 3$   | 2      | valid   | —          |
| C1    | Convolution     | 96      | $55 \times 55$   | $11 \times 11$ | 4      | valid   | ReLU       |
| In    | Input           | 3 (RGB) | $227 \times 227$ | —              | —      | —       | —          |

Figure 35: AlexNet architecture

A variant of AlexNet called *ZF Net* was developed by Matthew Zeiler and Rob Fergus and won the 2013 ILSVRC challenge. It is essentially AlexNet with a few tweaked hyperparameters (number of feature maps, kernel size, stride, etc.).

## GoogLeNet

The GoogLeNet architecture was developed by Google Research and it won the ILSVRC 2014 challenge. This great performance came in large part from the fact that the network was much deeper than previous CNNs. This was made possible by sub-networks called *inception modules*, which allow GoogLeNet to use parameters much more efficiently than previous architectures: GoogLeNet actually has 10 times fewer parameters than AlexNet (roughly 6 million instead of 60 million).

### Inception Module

The following figure shows the architecture of an inception module. The whole inception module can be considered as a convolutional layer on steroids, able to output feature maps that capture complex patterns at various scales. The module has convolutional layers with  $1 \times 1$  kernels for the following purposes.

- Although they cannot capture spatial patterns, they can capture patterns along the depth dimension (i.e., across channels).
- They are configured to output fewer feature maps than their inputs, so they serve as bottleneck layers, meaning they reduce dimensionality cutting down the computational cost and the number of parameters, speeding up training and improving generalization.
- Each pair of convolutional layers ( $[1 \times 1, 3 \times 3]$  and  $[1 \times 1, 5 \times 5]$ ) acts like a single

powerful convolutional layer, capable of capturing more complex patterns.

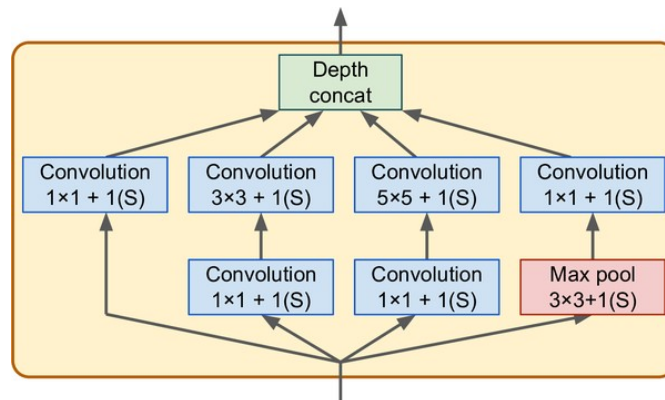


Figure 36: Inception module

The architecture of the GoogLeNet CNN is shown below. The number of feature maps output by each convolutional layer and each pooling layer is shown before the kernel size. GoogLeNet is actually **one tall stack, including nine inception modules** (the boxes with the spinning tops). The six numbers in the inception modules represent the number of feature maps output by each convolutional layer in the module (in the same order as in figure above). All the convolutional layers use the ReLU activation function.

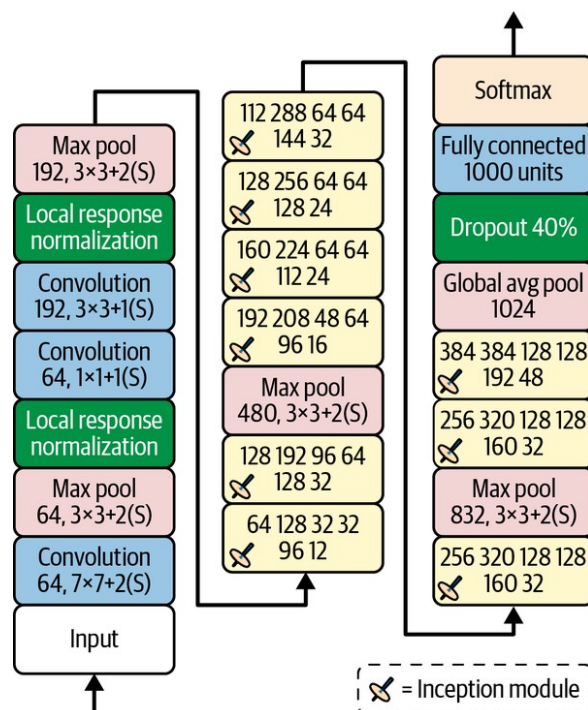


Figure 37: GoogLeNet architecture

### Discussion on the Network:

- The first two layers divide the image's height and width by 4 (so its area is divided by 16), to reduce the computational load. The first layer uses a large kernel size,  $7 \times 7$ , so that much of the information is preserved.
- Then the local response normalization layer ensures that the previous layers learn a wide variety of features.
- Next, a max pooling layer reduces the image height and width by 2, again to speed up computations.
- Then comes the CNN's backbone: a tall stack of nine inception modules, interleaved with a couple of max pooling layers to reduce dimensionality and speed up the net.
- Next, the global average pooling layer outputs the mean of each feature map dropping any remaining spatial information.
- The last layers are dropout for regularization, then a fully connected layer with 1,000 units (since there are 1,000 classes) and a softmax activation function to output estimated class probabilities.

### VGGNet

The runner-up in the ILSVRC 2014 challenge was VGGNet that had a very simple and classical architecture with 2 or 3 convolutional layers and a pooling layer, then again 2 or 3 convolutional layers and a pooling layer, and so on (reaching a total of 16 or 19 convolutional layers, depending on the VGG variant), plus a final dense network with 2 hidden layers and the output layer. It used small  $3 \times 3$  filters, but it had many of them.

### ResNet

ILSVRC 2015 challenge was won by a Residual Network (ResNet) that had a CNN composed of 152 layers confirming the general trend for computer vision models getting deeper and deeper, with fewer and fewer parameters. The key to being able to train such a deep network is to use skip connections (also called shortcut connections): the signal feeding into a layer is also added to the output of a layer located higher up the stack.

When training a neural network, the goal is to make it model a target function  $h(\mathbf{x})$ . If input  $\mathbf{x}$  is added to the output of the network (i.e., a skip connection is added), then the network will be forced to model  $f(\mathbf{x}) = h(\mathbf{x}) - \mathbf{x}$  rather than  $h(\mathbf{x})$ . This is called residual learning (see figure below).

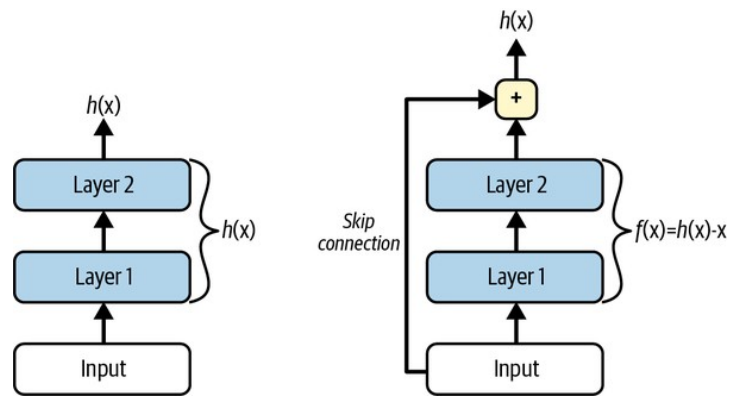


Figure 38: Residual learning

When a regular neural network is initialized, its weights are close to zero, so the network just outputs values close to zero. If you add a skip connection, the resulting network just outputs a copy of its inputs; in other words, it initially models the identity function. If the target function is fairly close to the identity function (which is often the case), this will speed up training considerably. Moreover, if many skip connections are added, the network can start making progress even if several layers have not started learning yet (see figure below). Thanks to skip connections, the signal can easily make its way across the whole network. The deep residual network can be seen as a stack of residual units (RUs), where each residual unit is a small neural network with a skip connection.

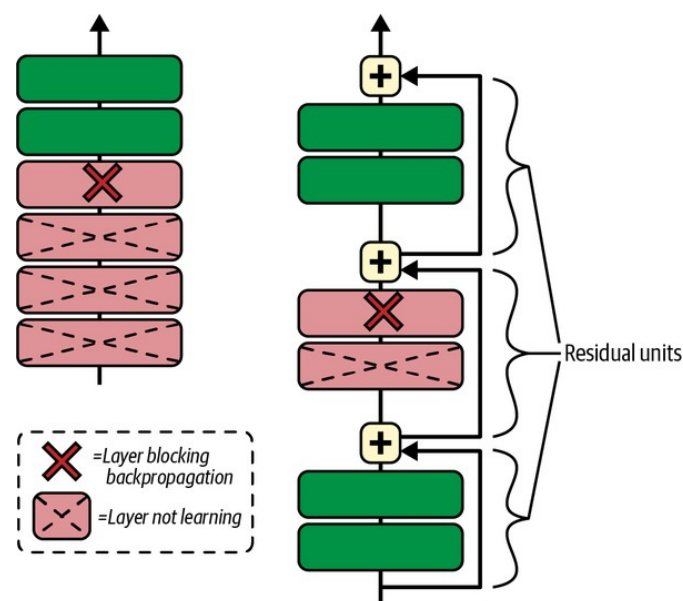


Figure 39: Regular deep neural network (left) and deep residual network (right)

ResNet's architecture (see figure below) is surprisingly simple. It starts and ends exactly like GoogLeNet (except without a dropout layer), and in between is just a very deep stack of residual units. Each residual unit is composed of two convolutional layers (and no pooling layer!), with batch normalization (BN) and ReLU activation, using  $3 \times 3$  kernels and preserving spatial dimensions (stride 1, "same" padding).



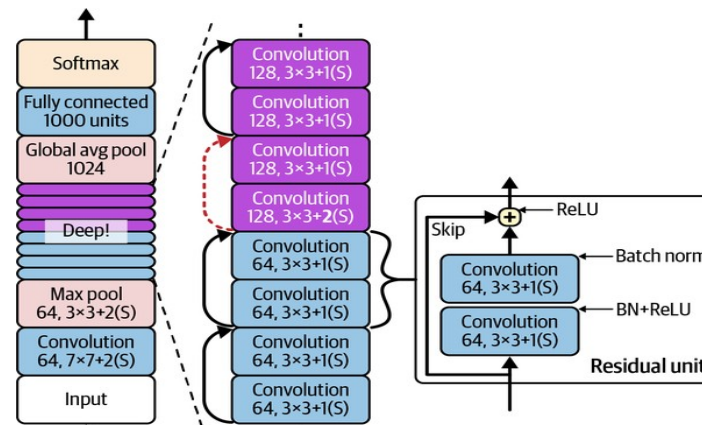


Figure 40: ResNet architecture

The number of feature maps is doubled every few residual units, at the same time as their height and width are halved (using a convolutional layer with stride 2). When this happens, the inputs cannot be added directly to the outputs of the residual unit because they don't have the same shape (for example, this problem affects the skip connection represented by the dashed arrow in previous figure). To solve this problem, the inputs are passed through a  $1 \times 1$  convolutional layer with stride 2 and the right number of output feature maps (see next figure).

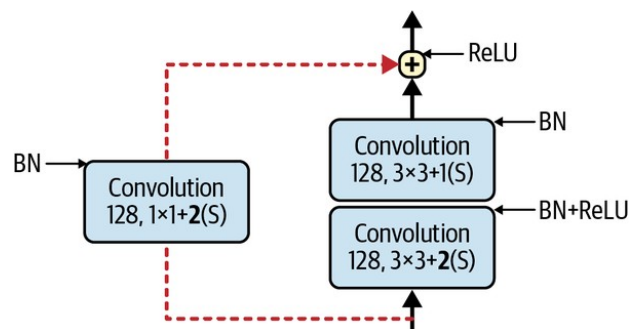


Figure 41: Skip connection when changing feature map size and depth

## Xception

Another variant of the GoogLeNet architecture is worth noting: Xception (which stands for *Extreme Inception*) was proposed in 2016 by François Chollet (the author of Keras) by merging the ideas of GoogLeNet and ResNet, but it replaces the inception modules with a special type of layer called a **depthwise separable convolution layer** (or separable convolution layer for short). While a regular convolutional layer uses filters that try to simultaneously capture spatial patterns (e.g., an oval) and cross-channel patterns (e.g., mouth + nose + eyes = face), a separable convolutional layer makes the strong assumption that spatial patterns and cross-channel patterns can be modeled separately (see figure below). Thus, it is composed of two parts: the first part applies a single spatial filter to each input feature map, then the second part looks exclusively for cross-channel patterns—it is just a regular convolutional layer with  $1 \times 1$  filters.



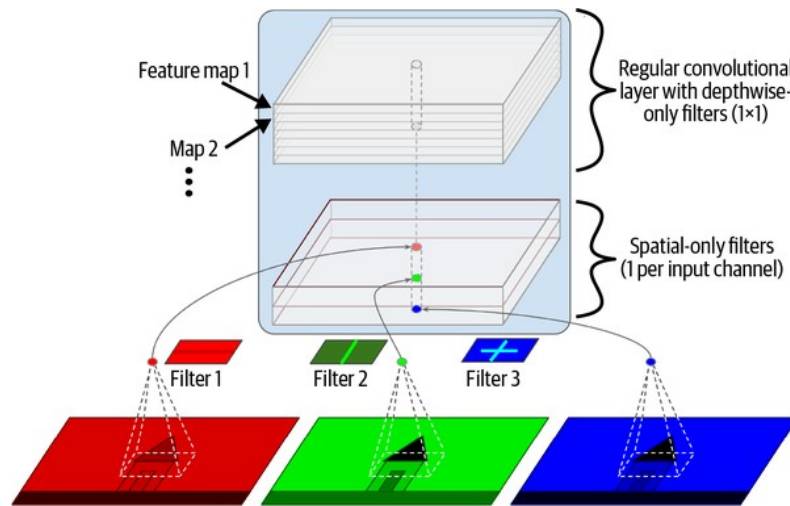


Figure 42: Depthwise separable convolutional layer

Since separable convolutional layers only have one spatial filter per input channel, Xception architecture starts with 2 regular convolutional layers, but then the rest of the architecture uses only separable convolutions (34 in all), plus a few max pooling layers and the usual final layers (a global average pooling layer and a dense output layer).

Separable convolutional layers use fewer parameters, less memory, and fewer computations than regular convolutional layers, and they often perform better. Usage of separable convolutional layers should be considered by default, except after layers with few channels (such as the input channel). In Keras, `SeparableConv2D` can be used instead of `Conv2D`: it's a drop-in replacement. Keras also offers a `DepthwiseConv2D` layer that implements the first part of a depthwise separable convolutional layer (i.e., applying one spatial filter per input feature map).

## SENet

The winning architecture in the ILSVRC 2017 challenge was the Squeeze-and-Excitation Network (SENet). This architecture extends existing architectures such as inception networks and ResNets, and boosts their performance. The extended versions of inception networks and ResNets are called *SE-Inception* and *SE-ResNet*, respectively. The boost comes from the fact that a SENet adds a small neural network, called an *SE block*, to every inception module or residual unit in the original architecture, as shown in the below figure.

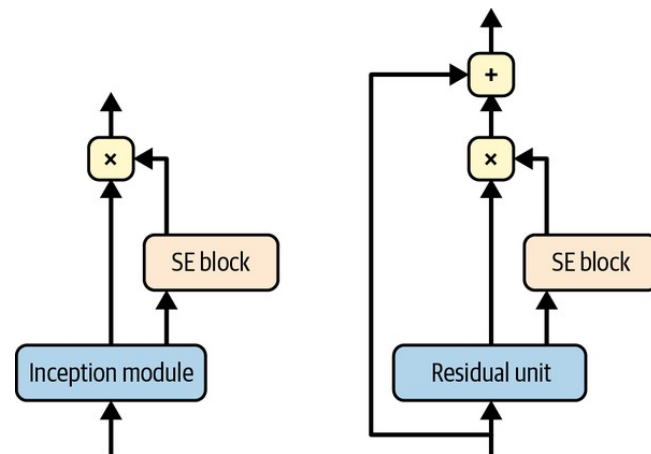


Figure 43: SE-Inception module (left) and SE-ResNet unit (right)

An SE block analyzes the output of the unit it is attached to, focusing exclusively on the depth dimension (it does not look for any spatial pattern), and it learns which features are usually most active together. It then uses this information to recalibrate the feature maps. An SE block is composed of just three layers: a global average pooling layer, a hidden dense layer using the ReLU activation function, and a dense output layer using the sigmoid activation function (see figure).

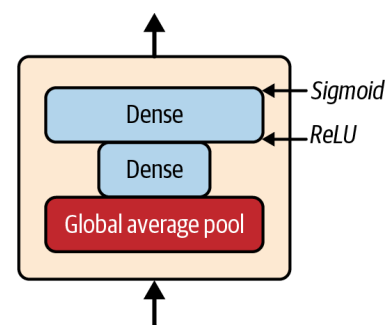


Figure 44: SE block architecture

As earlier, the global average pooling layer computes the mean activation for each feature map. The next layer is where the “squeeze” happens: this layer has significantly fewer neurons—typically several times fewer than the number of feature maps—resulting a compressed (low-dimensional) small vector (i.e., an embedding) representing the distribution of feature responses. This bottleneck step forces the SE block to learn a general representation of the feature combinations. Finally, the output layer takes the embedding and outputs a recalibration vector containing one number per feature map, each between 0 and 1. The feature maps are then multiplied by this recalibration vector, so irrelevant features (with a low recalibration score) get scaled down while relevant features (with a recalibration score close to 1) are left alone.

### Choosing the Right CNN Architecture

With so many CNN architectures, one is chosen depending on what matters for your project such as accuracy, model size (e.g., for deployment to a mobile device), inference speed on CPU and on GPU, etc. Table below lists the best pretrained models currently available in Keras, sorted by model size. For each model, the table shows the Keras class name to use (in the `tf.keras.applications` package), the model’s size in MB, the top-1 and top-5 validation accuracy on the ImageNet dataset, the number of parameters (millions), and the inference time on CPU and GPU in ms, using batches of 32 images on reasonably powerful hardware. For each column, the best value is highlighted. As

can be seen, larger models are generally more accurate, but not always; for example, EfficientNetB2 outperforms InceptionV3 both in size and accuracy. InceptionResNetV2 is fast on a CPU, and ResNet50V2 and ResNet101V2 are blazingly fast on a GPU.

| Class name        | Size (MB) | Top-1 acc | Top-5 acc | Params | CPU (ms) | GPU (ms) |
|-------------------|-----------|-----------|-----------|--------|----------|----------|
| MobileNetV2       | 14        | 71.3%     | 90.1%     | 3.5M   | 25.9     | 3.8      |
| MobileNet         | 16        | 70.4%     | 89.5%     | 4.3M   | 22.6     | 3.4      |
| NASNetMobile      | 23        | 74.4%     | 91.9%     | 5.3M   | 27.0     | 6.7      |
| EfficientNetB0    | 29        | 77.1%     | 93.3%     | 5.3M   | 46.0     | 4.9      |
| EfficientNetB1    | 31        | 79.1%     | 94.4%     | 7.9M   | 60.2     | 5.6      |
| EfficientNetB2    | 36        | 80.1%     | 94.9%     | 9.2M   | 80.8     | 6.5      |
| EfficientNetB3    | 48        | 81.6%     | 95.7%     | 12.3M  | 140.0    | 8.8      |
| EfficientNetB4    | 75        | 82.9%     | 96.4%     | 19.5M  | 308.3    | 15.1     |
| InceptionV3       | 92        | 77.9%     | 93.7%     | 23.9M  | 42.2     | 6.9      |
| ResNet50V2        | 98        | 76.0%     | 93.0%     | 25.6M  | 45.6     | 4.4      |
| EfficientNetB5    | 118       | 83.6%     | 96.7%     | 30.6M  | 579.2    | 25.3     |
| EfficientNetB6    | 166       | 84.0%     | 96.8%     | 43.3M  | 958.1    | 40.4     |
| ResNet101V2       | 171       | 77.2%     | 93.8%     | 44.7M  | 72.7     | 5.4      |
| InceptionResNetV2 | 215       | 80.3%     | 95.3%     | 55.9M  | 130.2    | 10.0     |
| EfficientNetB7    | 256       | 84.3%     | 97.0%     | 66.7M  | 1578.9   | 61.6     |

Table 3: Pretrained models available in Keras

## Exercises

1. What are the advantages of a CNN over a fully connected DNN for image classification?
2. Consider a CNN composed of three convolutional layers, each with  $3 \times 3$  kernels, a stride of 2, and "same" padding. The lowest layer outputs 100 feature maps, the middle one outputs 200, and the top one outputs 400. The input images are RGB images of  $200 \times 300$  pixels:
  - a) What is the total number of parameters in the CNN?
  - b) If we are using 32-bit floats, at least how much RAM will this network require when making a prediction for a single instance?
  - c) What about when training on a mini-batch of 50 images?
  - d) Why would memory requirement during inference be less as compared to the total memory requirement by all layers during model training.
3. If your GPU runs out of memory while training a CNN, what are five things you could try to solve the problem?

4. Why would you want to add a max pooling layer rather than a convolutional layer with the same stride?
5. When would you want to add a local response normalization layer?
6. Can you name the main innovations in AlexNet, as compared to LeNet-5? What about the main innovations in GoogLeNet, ResNet, SENet, Xception, and EfficientNet?

## Distributing TensorFlow Across Devices and Servers

Once a model makes reasonable predictions, it might need to be put in production. This could be as simple as running the model on a batch of data, and perhaps writing a script that runs this model every night. However, this model might need to be used on live data, in which case the model could be **wrapped in a web service** so that the model can be queried at any time using a simple **REST API** (or some other protocol such as Google Remote Procedure Call or **gRPC**). But as time passes, the model might also need to be **retrained regularly** on fresh data and pushed the updated version to production. Other considerations such as handling **model versioning**, gracefully transition from one model to the next, possibly rolling back to the previous model in case of problems, and perhaps running multiple different models in parallel to perform **A/B experiments**. If the product becomes successful, the service may start getting a **large number of queries per second** (QPS), and it must **scale up to support the load**. A great solution to scale up your service is to use **TF Serving**, either on own hardware infrastructure or via a cloud service such as **Google Vertex AI**. It will take care of efficiently serving the model, handle graceful model transitions, and more. The cloud platforms also come with many extra features such as powerful monitoring tools.

If a **compute-intensive model** needs to be trained on a **lot of training data**, then **training time** may be prohibitively long, and it could be a showstopper for **performing multiple experiments**. One way to **speed up** training is to use **hardware accelerators such as GPUs or TPUs** or go even faster, model can be trained **across multiple machines**, each equipped with multiple hardware accelerators. TensorFlow is one of the deep learning libraries that provides simple yet powerful **distribution strategies API** that makes this easy.

### Serving a TensorFlow Model

If the trained model needs to be used on live data, then it is preferable to **wrap the model in a small service** whose sole role is to make predictions against queries (e.g., via a **REST** or **gRPC** API). This decouples the model from the rest of the infrastructure, making it possible to easily switch model versions or scale the service up as needed (independently from the rest of your infrastructure), perform A/B experiments, and ensure that all the software components rely on the same model versions. It also simplifies testing and development, and more. Microservice can be created using any technology (e.g., using the **Flask library**), but reinventing the wheel when just **TF Serving** can be used may not make sense.

#### *Using TensorFlow Serving*

TF Serving is a very efficient, battle-tested model server, written in C++. It can sustain a **high load**, serve **multiple versions** of your models and **watch a model repository** to automatically deploy the latest versions, and more (see figure below).

The followings shows how to use TF Serving to deploy a model trained on MNIST using Keras. The first thing that needs to be done is to export this model to the **SavedModel** format.

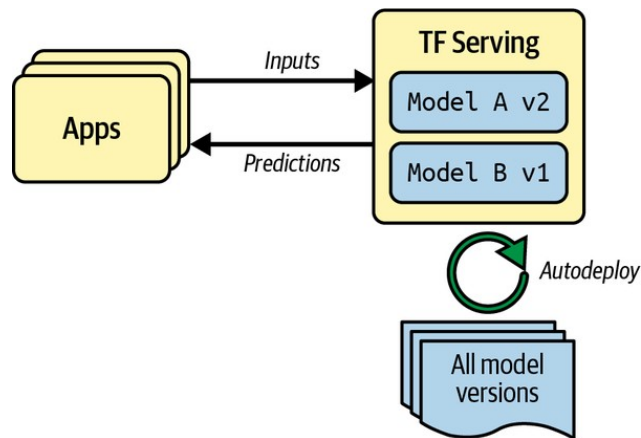


Figure 45: TF Serving can serve multiple models and automatically deploy the latest version of each model

## Exporting SavedModels

To save the model function `model.save()` is called. Now to version the model, a `subdirectory` for each model needs to be created.

```
from pathlib import Path
import tensorflow as tf

X_train, X_valid, X_test = [...] # load and split the MNIST dataset
model = [...] # build & train an MNIST model (also handles image preprocessing)
model_name = "my_mnist_model"
model_version = "0001"
model_path = Path(model_name) / model_version
model.save(model_path, save_format="tf")
```

It's usually a good idea to include all the preprocessing layers in the final model you export so that it can ingest data in its natural form once it is deployed to production. This avoids having to take care of preprocessing separately within the application that uses the model. Bundling the preprocessing steps within the model also makes it simpler to update them later on and limits the risk of mismatch between a model and the preprocessing steps it requires.

TensorFlow comes with a small `saved_model_cli` command-line interface to `inspect SavedModels`. The following inspects the current model.

```
$ saved_model_cli show --dir my_mnist_model/0001
The given SavedModel contains the following tag-sets:
'serve'
```

A `SavedModel` contains one or more `metagraphs`. A metagraph is a `computation graph plus some function signature definitions, including their input and output names, types, and shapes`. Each metagraph is identified by a `set of tags`. For example, if a metagraph containing the full computation graph, including the training operations is required: that can be `tagged as "train"`. And for metagraph containing a pruned computation graph with only the prediction operations, including some GPU-specific operations: can be `tagged as "serve", "gpu"`. When a Keras model is saved using its `save()` method, it saves a single metagraph tagged as "serve". The following inspects this "serve" tag set.

```
$ saved_model_cli show --dir 0001/my_mnist_model --tag_set serve
The given SavedModel MetaGraphDef contains SignatureDefs with these keys:
SignatureDef key: "__saved_model_init_op"
SignatureDef key: "serving_default"
```

This metagraph contains two signature definitions: an initialization function called "\_\_saved\_model\_init\_op", which need not be worried about, and a default serving function called "serving\_default". When saving a Keras model, the default serving function is the model's `call()` method, which makes predictions. More details about this serving function is shown below.

```
$ saved_model_cli show --dir 0001/my_mnist_model --tag_set serve \
--signature_def serving_default
The given SavedModel SignatureDef contains the following input(s):
  inputs['flatten_input'] tensor_info:
    dtype: DT_UINT8
    shape: (-1, 28, 28)
    name: serving_default_flatten_input:0
The given SavedModel SignatureDef contains the following output(s):
  outputs['dense_1'] tensor_info:
    dtype: DT_FLOAT
    shape: (-1, 10)
    name: StatefulPartitionedCall:0
Method name is: tensorflow/serving/predict
```

The function's input is named "flatten\_input", and the output is named "dense\_1". These correspond to the Keras model's input and output layer names. The type and shape of the input and output data can also be referred.

## Querying TF Serving through the REST API

To create the query, it must contain the name of the **function signature** to be called, and of course the **input data**. Since the request must use the **JSON format**, the input images need to be converted from a NumPy array to a Python list.

```
import json

X_new = X_test[:3] # pretend there are 3 new digit images to classify
request_json = json.dumps({
    "signature_name": "serving_default",
    "instances": X_new.tolist(),
})
```

As JSON format is **100% text-based**, the request string looks like this:

```
request_json
'{"signature_name": "serving_default", "instances": [[[0, 0, 0, 0, ... ]]]}'
```

To send this request to TF Serving via an **HTTP POST** request, `requests` library can be used.

```
import requests
```

```
server_url = "http://localhost:8501/v1/models/my_mnist_model:predict"
response = requests.post(server_url, data=request_json)
response.raise_for_status() # raise an exception in case of error
response = response.json()
```

If all goes well, the response should be a dictionary containing a single "predictions" key. The corresponding value is the Python list of predictions. It gets converted to a NumPy array and rounded the floats to contain second decimal.

```
import numpy as np

y_proba = np.array(response["predictions"])
y_proba.round(2)
array([[0. , 0. , 0. , 0. , 0. , 0. , 0. , 1. , 0. , 0. ],
       [0. , 0. , 0.99, 0.01, 0. , 0. , 0. , 0. , 0. , 0. ],
       [0. , 0.97, 0.01, 0. , 0. , 0. , 0. , 0.01, 0. , 0. ]])
```

The REST API is simple, and it works well when the input and output data are not too large. However, it is based on JSON, which is text-based and fairly verbose. For the above example, the NumPy array was converted to Python list, and every float ended up represented as a string. This is very inefficient, both in terms of serialization/deserialization time and in terms of payload size. This will result in high latency and bandwidth usage when transferring large NumPy arrays. So, in these cases, gRPC (that uses a compact binary format and an efficient communication protocol based on HTTP/2 framing) can be used instead.

## Querying TF Serving through the gRPC API

The gRPC API expects a serialized PredictRequest protocol buffer as input, and it outputs a serialized PredictResponse protocol buffer. These protocol buffers are part of the tensorflow-serving-api library and it works as follows.

```
from tensorflow_serving.apis.predict_pb2 import PredictRequest

request = PredictRequest()
request.model_spec.name = model_name
request.model_spec.signature_name = "serving_default"
input_name = model.input_names[0] # == "flatten_input"
request.inputs[input_name].CopyFrom(tf.make_tensor_proto(X_new))
```

This code creates a PredictRequest protocol buffer and fills in the required fields, including the model name (defined earlier), the signature name of the function to be called, and finally the input data, in the form of a Tensor protocol buffer. The tf.make\_tensor\_proto() function creates a Tensor protocol buffer based on the given tensor or NumPy array, in this case X\_new.

Next, the request to sent to the server through grpcio library and its response is received.

```
import grpc
from tensorflow_serving.apis import prediction_service_pb2_grpc

channel = grpc.insecure_channel('localhost:8500')
predict_service = prediction_service_pb2_grpc.PredictionServiceStub(channel)
```



```
response = predict_service.Predict(request, timeout=10.0)
```

After the imports, a gRPC (insecure) communication channel is created to *localhost* on TCP port 8500, followed by creating a gRPC service over this channel and using it to send a synchronous request (blocking call) with a 10-second timeout.

Next, the `PredictResponse` protocol buffer gets converted to a tensor.

```
output_name = model.output_names[0]      # == "dense_1"
outputs_proto = response.outputs[output_name]
y_proba = tf.make_ndarray(outputs_proto)
```

Printing `y_proba.round(2)`, will get the exact same estimated class probabilities as earlier. This is how TensorFlow model can be accessed remotely, using either REST or gRPC.

## Deploying a New Model Version

The following code exports a new model version into a `SavedModel` to folder `my_mnist_model/0002` directory.

```
model = [...] # build and train a new MNIST model version

model_version = "0002"
model_path = Path(model_name) / model_version
model.save(model_path, save_format="tf")
```

At regular intervals (the delay is configurable), **TF Serving checks the model directory** for new model versions. If it finds one, it **automatically handles the transition gracefully**: by default, it answers **pending requests** (if any) with the previous model version, while handling **new requests** with the new version. As soon as every pending request has been answered, the **previous model version is unloaded**. This approach offers a smooth transition, but it may use too much RAM—especially GPU RAM, which is generally the most limited. In this case, TF Serving can be configured so that it handles all pending requests with the previous model version and unloads it before loading and using the new model version avoiding having two model versions loaded at the same time, but the **service will be unavailable** for a short period. Moreover, if it is found that version 2 does not work as expected, then rolling back to version 1 is as simple as removing the `my_mnist_model/0002` directory.

Another great feature of TF Serving is its automatic **batching capability**. If it is activated using the `--enable_batching` option upon startup and TF Serving receives multiple requests within a short period of time (the delay is configurable), it will automatically batch them together before using the model. This offers a significant performance boost by leveraging the power of the GPU. Once the model returns the predictions, TF Serving dispatches each prediction to the right client.

If many queries per second is expected, **deploying TF Serving on multiple servers and load-balancing the queries** (see figure below) can be considered. This will require deploying and managing many **TF Serving containers** across these servers. One way to handle that is to use a tool such as **Kubernetes** which is container orchestrator across many servers. If owning, maintaining,

and upgrading infrastructure are not intended, then virtual machines on a cloud platform such as Amazon AWS, Microsoft Azure, Google Cloud Platform, or some other platform as a service (PaaS) offering can be considered. Managing all the virtual machines, handling container orchestration (even with the help of Kubernetes), taking care of TF Serving configuration, tuning and monitoring—all of this can be taken care of by some service providers such as Vertex AI that supports TPU, TensorFlow 2, Scikit-Learn, and XGBoost, and offers other AI services. There are several other providers such as Amazon AWS SageMaker and Microsoft AI Platform, that also offer serving TensorFlow models as well.

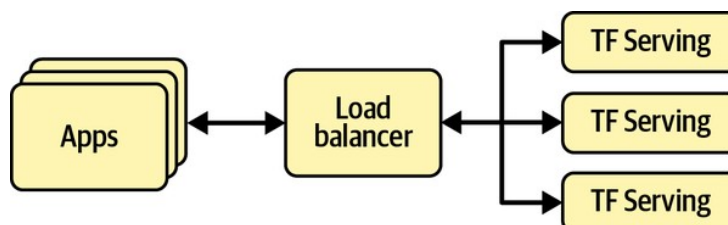


Figure 46: Scaling up TF Serving with load balancing

## Using GPUs to Speed Up Computations

Several techniques such as better weight initialization, sophisticated optimizers, and so on can considerably speed up training. But even with all of these techniques, training a large neural network on a single machine with a single CPU can take hours, days, or even weeks, depending on the task. Thanks to GPUs, this training time can be reduced down to minutes or hours. Not only does this save an enormous amount of time, but it also means that experiments with various models can be performed much more easily, and models can be retrained frequently on fresh data. The followings will discuss about the **distribution of computations across the CPU and multiple GPU devices** on a single machine (see figure below), then on how to distribute computations across multiple servers.

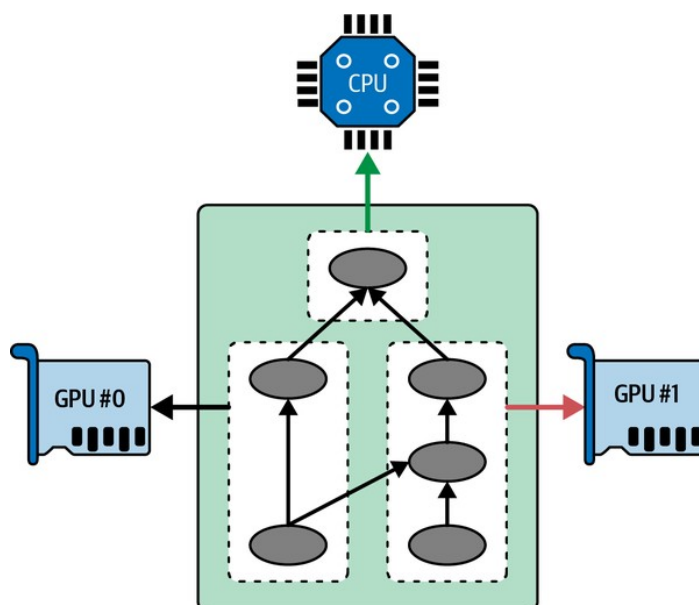


Figure 47: Executing a TensorFlow graph across multiple devices in parallel

## The GPU in the Local Machine

If data is not intended to be copied into the cloud then GPU in the local machine can be considered to train machine learning models. Few of the important specifications of the GPUs are **RAM** (e.g., typically at least 10 GB for image processing or NLP), **bandwidth** (i.e., how fast data can be sent into and taken out of the GPU), the **number of cores**, the **cooling system**, etc. Presently TensorFlow only supports Nvidia cards with CUDA Compute Capability 3.5+ (as well as Google's TPUs, of course).

Appropriate Nvidia drivers and several Nvidia libraries need to be installed for Nvidia GPU card to be used. These include the **Compute Unified Device Architecture library (CUDA) Toolkit**, which allows developers to use CUDA-enabled GPUs for all sorts of computations (not just graphics acceleration), and the **CUDA Deep Neural Network library (cuDNN)**, a GPU-accelerated library of common DNN computations such as activation layers, normalization, forward and backward convolutions, and pooling. TensorFlow uses CUDA and cuDNN to control the GPU cards and accelerate computations (see figure below).

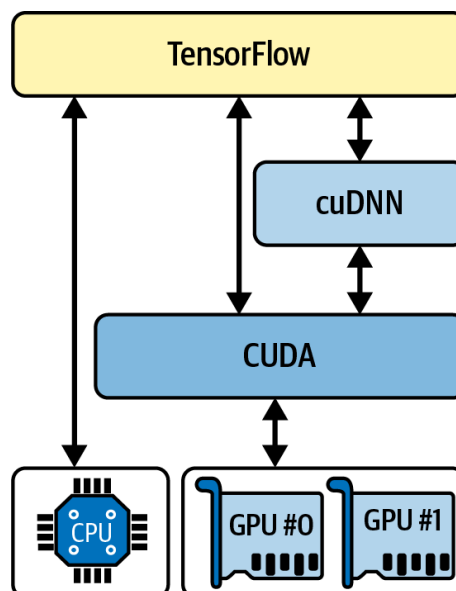


Figure 48: TensorFlow uses CUDA and cuDNN to control GPUs and boost DNNs

## Managing the GPU RAM

By default, TensorFlow automatically **grabs almost all the RAM** in all available GPUs the first time a computation takes place. It does this to limit GPU RAM fragmentation. Starting a second TensorFlow program (or any program that requires the GPU), it may run out of RAM quickly. To run multiple programs (e.g., to train two different models in parallel on the same machine), then you the GPU RAM can be split between these processes more evenly.

Similarly, for a machine multiple GPU cards, **each GPU can be assigned to a single process**. To do this, **CUDA\_VISIBLE\_DEVICES environment variable** should be set so that each process only sees the appropriate GPU card(s). The **CUDA\_DEVICE\_ORDER environment variable** should also be set to PCI\_BUS\_ID to ensure that each ID always refers to the same GPU card. For example, in

case of a machine with four GPU cards, two programs can be started, assigning two GPUs to each of them, by executing commands like the following in two separate terminal windows.

```
$ CUDA_DEVICE_ORDER=PCI_BUS_ID CUDA_VISIBLE_DEVICES=0,1 python3 program_1.py  
# and in another terminal:  
$ CUDA_DEVICE_ORDER=PCI_BUS_ID CUDA_VISIBLE_DEVICES=3,2 python3 program_2.py
```

Program 1 will then only see GPU cards 0 and 1, named `"/gpu:0"` and `"/gpu:1"`, respectively, in TensorFlow, and program 2 will only see GPU cards 2 and 3, named `"/gpu:1"` and `"/gpu:0"`, respectively (see figure below). Of course, these environment variables can also be defined in Python by setting `os.environ["CUDA_DEVICE_ORDER"]` and `os.environ["CUDA_VISIBLE_DEVICES"]` before using TensorFlow.

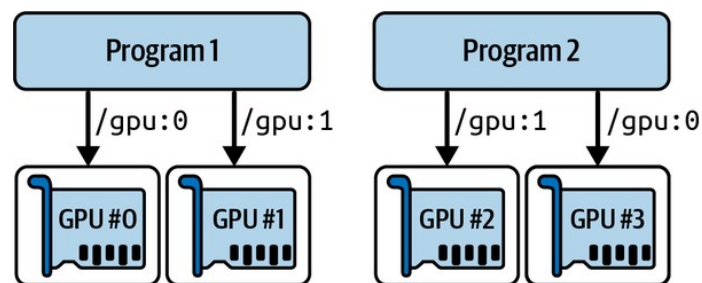


Figure 49: Each program gets two GPUs

Another option is to tell TensorFlow to grab only a **specific amount of GPU RAM**. This must be done immediately after importing TensorFlow. For example, to make TensorFlow grab only 2 GiB of RAM on each GPU, a logical GPU device (sometimes called a virtual GPU device) must be created for each physical GPU device and set its memory limit to 2 GiB (i.e., 2,048 MiB).

```
for gpu in physical_gpus:  
    tf.config.set_logical_device_configuration(  
        gpu,  
        [tf.config.LogicalDeviceConfiguration(memory_limit=2048)]  
    )
```

For a machine with four GPUs, each with at least 4 GiB of RAM, two programs like this one can run in parallel, each using all four GPU cards (see figure below). If command `nvidia-smi` is executed while both programs are running, it can be seen that each process holds 2 GiB of RAM on each card.

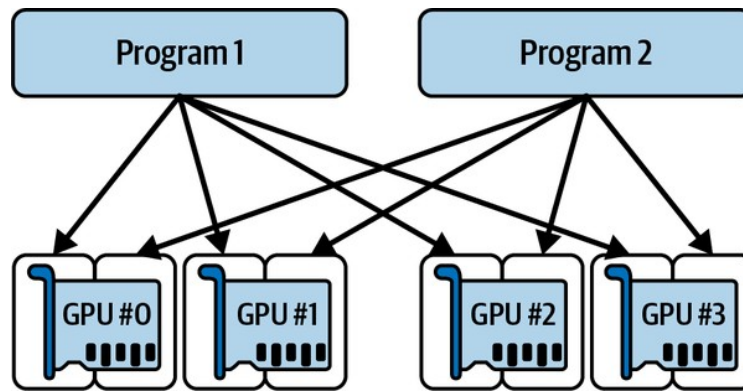


Figure 50: Each program gets all four GPUs, but with only 2 GiB of RAM on each GPU

Another way to do this is to set the `TF_FORCE_GPU_ALLOW_GROWTH` environment variable to true. With this option, TensorFlow will never release memory once it has grabbed it (again, to avoid memory fragmentation), except of course when the program ends. It can be harder to guarantee deterministic behavior using this option (e.g., one program may crash because another program's memory usage went through the roof). However, there are some cases where it is very useful: for example, while using a machine to run multiple Jupyter notebooks, several of which use TensorFlow. The `TF_FORCE_GPU_ALLOW_GROWTH` environment variable is set to true in Colab runtimes.

Lastly, in some cases where a GPU might need to be `split into two or more logical devices`. For example, this is useful if there is one physical GPU — like in a Colab runtime — but a multi-GPU algorithm needs to be tested. The following code splits GPU #0 into two logical devices, with 2 GiB of RAM each (again, this must be done immediately after importing TensorFlow).

```
tf.config.set_logical_device_configuration(
    physical_gpus[0],
    [tf.config.LogicalDeviceConfiguration(memory_limit=2048),
     tf.config.LogicalDeviceConfiguration(memory_limit=2048)]
)
```

These two logical devices are called `"/gpu:0"` and `"/gpu:1"`, and these can be used if they were two normal GPUs. These logical devices can be listed like this.

```
logical_gpus = tf.config.list_logical_devices("GPU")
logical_gpus
[LogicalDevice(name='/device:GPU:0', device_type='GPU'),
 LogicalDevice(name='/device:GPU:1', device_type='GPU')]
```

The CPU is always treated as a single device (`"/cpu:0"`), even the machine has multiple CPU cores. Any operation placed on the CPU may run in parallel across multiple cores if it has a multithreaded kernel.

## Placing Operations and Variables on Devices

Keras and `tf.data` generally do a good job of placing operations and variables where they belong, but operations and variables can also be placed manually on each device based on the following considerations.

- Data preprocessing operations to be performed on the CPU, and neural network operations on the GPUs.
- Avoiding unnecessary data transfers into and out of the GPUs because GPUs usually have a fairly limited communication bandwidth.
- Considering variables to be placed on CPU RAM instead of GPU RAM if a variable is not needed in the next few training steps because GPU RAM is expensive and limited resource as compared to CPU RAM which is relatively much cheaper and available in plenty, and upgrading GPU RAM is not as straightforward as it could be for CPU RAM.

By default, all variables and all operations will be placed on the **first GPU** (the one named `"/gpu:0"`), except for variables and operations that don't have a GPU kernel; these are placed on the CPU (always named `"/cpu:0"`). A tensor or variable's device attribute tells you which device it was placed on.

```
a = tf.Variable([1., 2., 3.]) # float32 variable goes to the GPU
a.device
'/job:localhost/replica:0/task:0/device:GPU:0'

b = tf.Variable([1, 2, 3]) # int32 variable goes to the CPU
b.device
'/job:localhost/replica:0/task:0/device:CPU:0'
```

If an operation needs to be placed on a different device than the default one, `tf.device()` context can be used.

```
with tf.device("/cpu:0"):
    c = tf.Variable([1., 2., 3.])
    ...

c.device
'/job:localhost/replica:0/task:0/device:CPU:0'
```

If an operation or variable is explicitly tried to be placed on a device that does not exist or for which there is no kernel, then TensorFlow will silently fallback to the device it would have chosen by default. This is useful when the same code is run on different machines that don't have the same number of GPUs. However, `tf.config.set_soft_device_placement(False)` can be set if an exception is preferred to be raised.

## Parallel Execution Across Multiple Devices

One of the benefits of using TF functions is **parallelism**. When TensorFlow runs a TF function, it starts by **analyzing its graph** to find the list of operations that need to be evaluated, and it counts how many **dependencies** each of them has. TensorFlow then adds each **operation** with zero

dependencies (i.e., each source operation) to the **evaluation queue** of this operation's device (see figure below). Once an operation has been evaluated, the **dependency counter** of each operation that depends on it is **decremented**. Once an operation's dependency counter **reaches zero**, it is pushed to the evaluation queue of its device. And once all the outputs have been computed, they are returned.

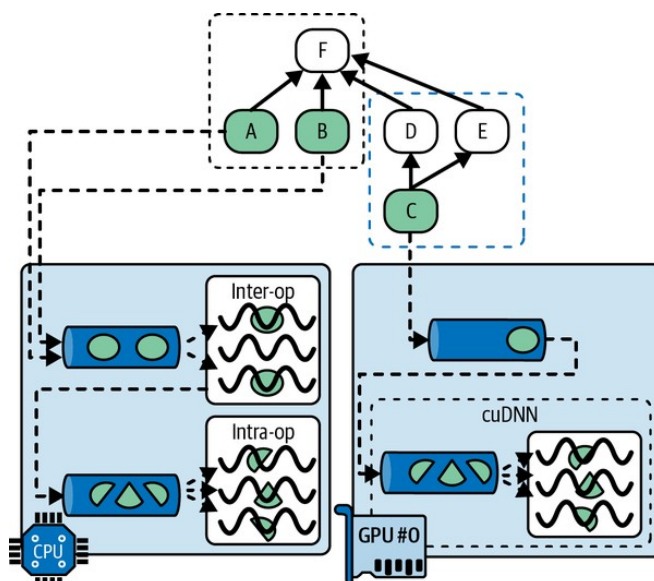


Figure 51: Parallelized execution of a TensorFlow graph

Operations in the CPU's evaluation queue are dispatched to a **thread pool** called the **inter-op thread pool**. If the CPU has multiple cores, then these operations will effectively be evaluated in parallel. Some operations have multithreaded CPU kernels: these kernels split their tasks into multiple sub-operations, which are placed in **another evaluation queue** and dispatched to a **second thread pool** called the **intra-op thread pool** (shared by all multithreaded CPU kernels). In short, multiple operations and sub-operations may be evaluated in parallel on different CPU cores.

For the GPU, things are a bit simpler. Operations in a **GPU's evaluation queue are evaluated sequentially**. However, most operations have multithreaded GPU kernels, typically implemented by libraries that TensorFlow depends on, such as CUDA and cuDNN. These implementations have their own thread pools, and they typically **exploit as many GPU threads as they can** (which is the reason why there is no need for an inter-op thread pool in GPUs: each operation already floods most GPU threads).

For example, in the above figure, operations A, B, and C are source ops, so they can immediately be evaluated. Operations A and B are placed on the CPU, so they are sent to the CPU's evaluation queue, then they are dispatched to the inter-op thread pool and immediately evaluated in parallel. Operation A happens to have a multithreaded kernel; its computations are split into three parts, which are executed in parallel by the intra-op thread pool. Operation C goes to GPU #0's evaluation queue, and its GPU kernel uses cuDNN, which manages its own intra-op thread pool and runs the operation across many GPU threads in parallel. Suppose C finishes first. The dependency counters of D and E are decremented and they reach 0, so both operations are pushed to GPU #0's evaluation queue, and they are executed sequentially. Note that C only gets evaluated once, even though both D and E depend on it. Suppose B finishes next. Then F's dependency counter is decremented from 4



to 3, and since that's not 0, it does not run yet. Once A, D, and E are finished, then F's dependency counter reaches 0, and it is pushed to the CPU's evaluation queue and evaluated. Finally, TensorFlow returns the requested outputs.

The number of threads in the inter-op thread pool and intra-op thread pool can be set by calling `tf.config.threading.set_inter_op_parallelism_threads()` and `tf.config.threading.set_intra_op_parallelism_threads()`, respectively. This is useful TensorFlow should not use all the CPU cores or single-threaded execution is expected.

## Training Models Across Multiple Devices

There are **two main approaches** to train a single model across multiple devices: **model parallelism**, where the model is split across the devices, and **data parallelism**, where the model is replicated across every device, and each replica is trained on a different subset of the data. These options are discussed below.

### Model Parallelism

To train a single neural network across multiple devices would require **chopping the model into separate chunks** and running each chunk on a different devices. Unfortunately, such model parallelism turns out to be pretty **tricky**, and its **effectiveness** really depends on the architecture of the neural network. For fully connected networks, there is generally **not much to be gained** from this approach (for example, by **splitting the model horizontally** (see figure below) and placing each layer on a different device), because each layer **needs to wait** for the output of the previous layer before it can do anything. Other option would be to **slicing it vertically**—for example, with the left half of each layer on one device, and the right part on another device. This is **slightly better**, since both halves of each layer can indeed work in parallel, but the problem is that each half of the next layer requires the output of both halves, so there will be a lot of **cross-device communication** (represented by the dashed arrows). This is likely to completely **cancel out the benefit** of the parallel computation, since cross-device communication is slow (and even more so when the devices are located on different machines).

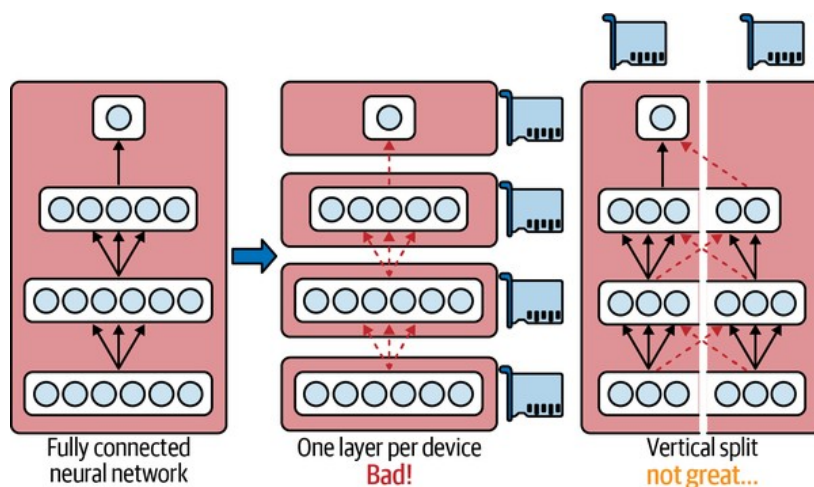


Figure 52: Splitting a fully connected neural network



Some neural network architectures, such as convolutional neural networks, contain layers that are only **partially connected to the lower layers**, so it is much **easier to distribute chunks** across devices in an efficient way (see figure below).

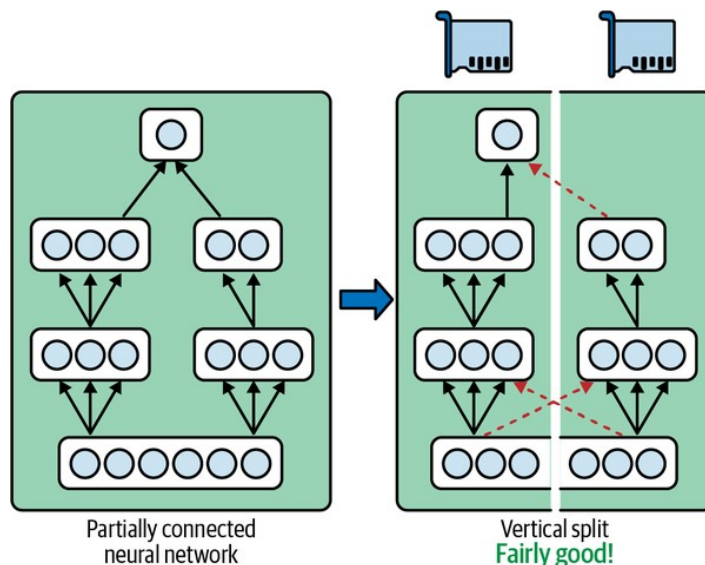


Figure 53: Splitting a partially connected neural network

Deep recurrent neural networks can be split a bit more efficiently across multiple GPUs. If you split the network horizontally by placing each layer on a different device, and feed the network with an input sequence to process, then at the first time step only one device will be active (working on the sequence's first value), at the second step two will be active (the second layer will be handling the output of the first layer for the first value, while the first layer will be handling the second value), and by the time the signal propagates to the output layer, all devices will be active simultaneously (see figure below). There is still a lot of cross-device communication going on, but since each cell may be fairly complex, the benefit of running multiple cells in parallel may (in theory) outweigh the communication penalty. However, in practice a regular stack of LSTM layers running on a single GPU actually runs much faster.

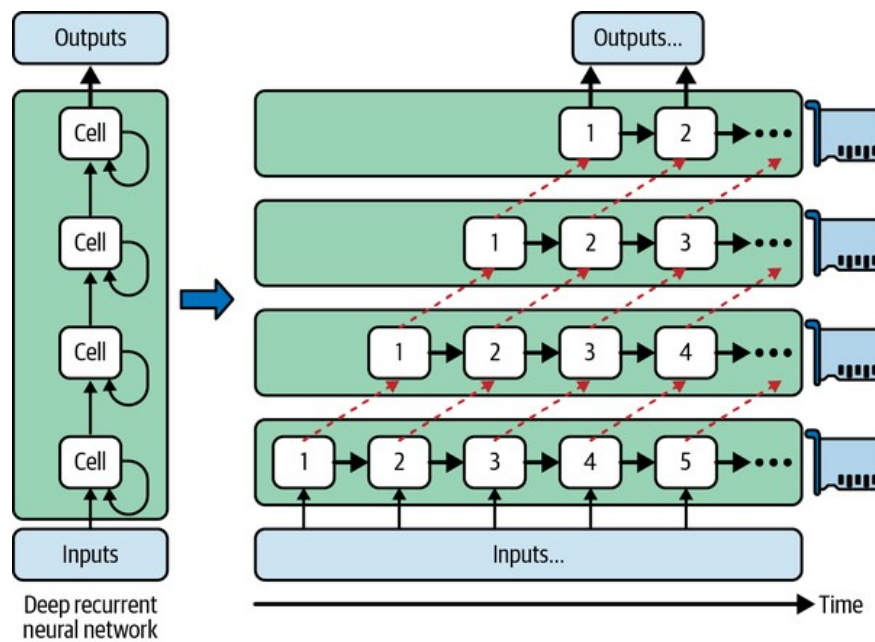


Figure 54: Splitting a deep recurrent neural network

In short, model parallelism may speed up running or training some types of neural networks, but not all, and it requires special care and tuning, such as making sure that devices that need to communicate the most run on the same machine.

### Data Parallelism

Another way to parallelize the training of a neural network is to replicate it on every device and run each training step simultaneously on all replicas, using a **different mini-batch** for each. The **gradients** computed by each replica are then **averaged**, and the result is used to **update the model parameters**. This is called *data parallelism*, or sometimes single program, multiple data (SPMD). Few of the important variants of this idea are discussed below.

### Data parallelism using the mirrored strategy

The simplest approach is to completely **mirror all the model parameters** across all the GPUs and always apply the exact same parameter updates on every GPU. This way, all replicas always remain perfectly identical. This is called the **mirrored strategy**, and it turns out to be quite efficient, especially when using a single machine (see figure below).

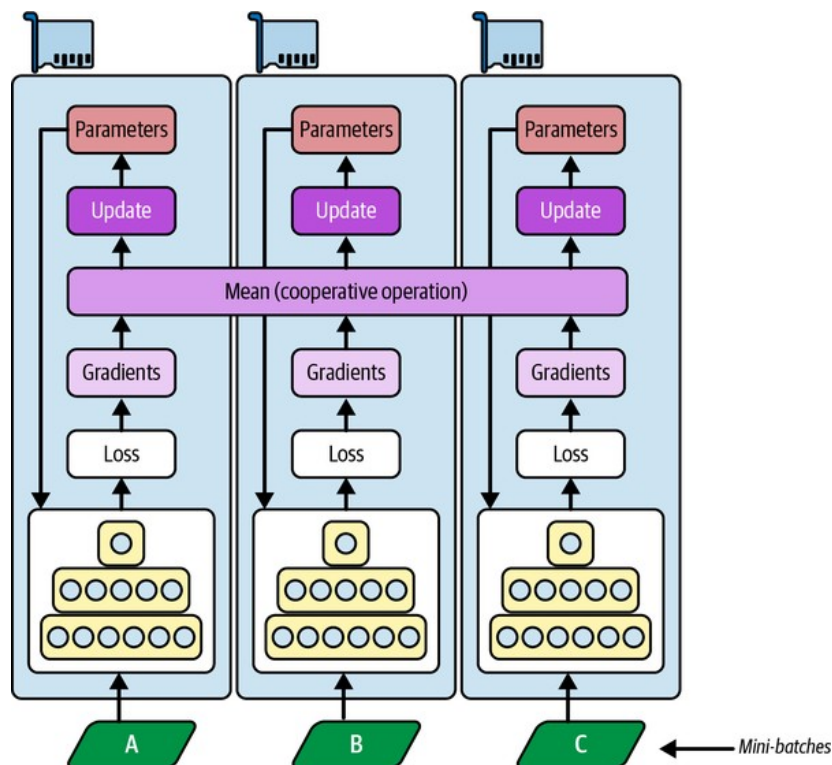


Figure 55: Data parallelism using the mirrored strategy

The tricky part when using this approach is to efficiently compute the mean of all the gradients from all the GPUs and distribute the result across all the GPUs. This can be done using an AllReduce algorithm, a class of algorithms where multiple nodes collaborate to efficiently perform a reduce operation (such as computing the mean, sum, and max), while ensuring that all nodes obtain the same final result.

### Data parallelism with centralized parameters

Another approach is to store the model parameters outside of the GPU devices performing the computations (called workers); for example, on the CPU (see figure below). In a distributed setup, all the parameters are placed on one or more CPU-only servers called parameter servers, whose only role is to host and update the parameters.

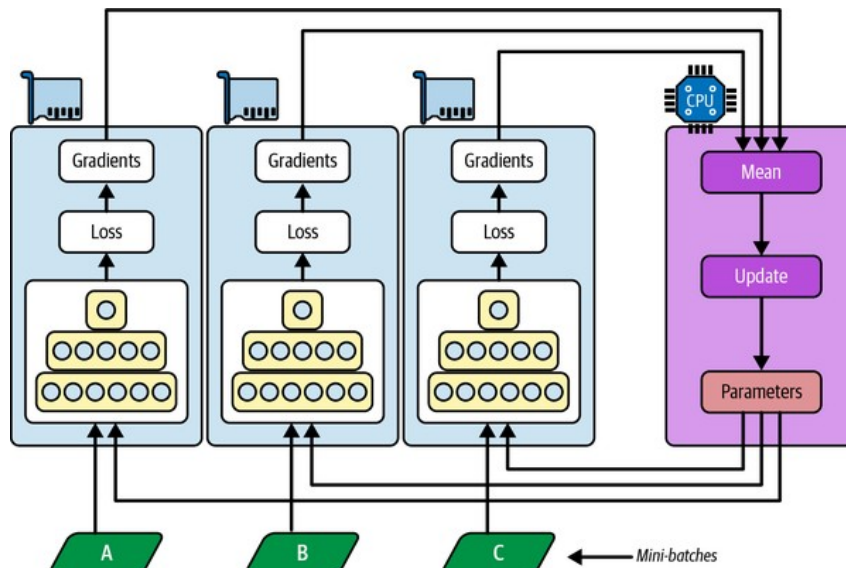


Figure 56: Data parallelism with centralized parameters

Whereas the mirrored strategy imposes synchronous weight updates across all GPUs, this centralized approach allows either **synchronous or asynchronous updates**.

**Synchronous updates:** With synchronous updates, the **aggregator waits until all gradients are available** before it computes the **average gradients** and passes them to the optimizer, which will update the model parameters. Once a replica has finished computing its gradients, it must wait for the parameters to be updated before it can proceed to the next mini-batch. The downside is that some devices may be **slower** than others, so the fast devices will have to wait for the slow ones at every step, making the whole process as slow as the slowest device. Moreover, the parameters will be copied to every device almost at the same time (immediately after the gradients are applied), which may saturate the parameter servers' bandwidth.

**Asynchronous updates:** With asynchronous updates, whenever a replica has finished computing the gradients, the gradients are immediately used to update the model parameters. There is **no aggregation** (it removes the "mean" step in from the above figure) and **no synchronization**. Replicas work independently of the other replicas. Since there is no waiting for the other replicas, this approach runs **more training steps per minute**. Moreover, although the parameters still need to be copied to every device at every step, this happens at different times for each replica, so the risk of bandwidth saturation is reduced. It works reasonably well in practice. Indeed, by the time a replica has finished computing the gradients based on some parameter values, these parameters will have been updated several times by other replicas (on average  $N - 1$  times, if there are  $N$  replicas), and there is no guarantee that the computed gradients will still be pointing in the right direction (see figure below). When gradients are severely out of date, they are called **stale gradients: they can slow down convergence, introducing noise and wobble effects** (the learning curve may contain temporary oscillations), or they can even make the training algorithm **diverge**.

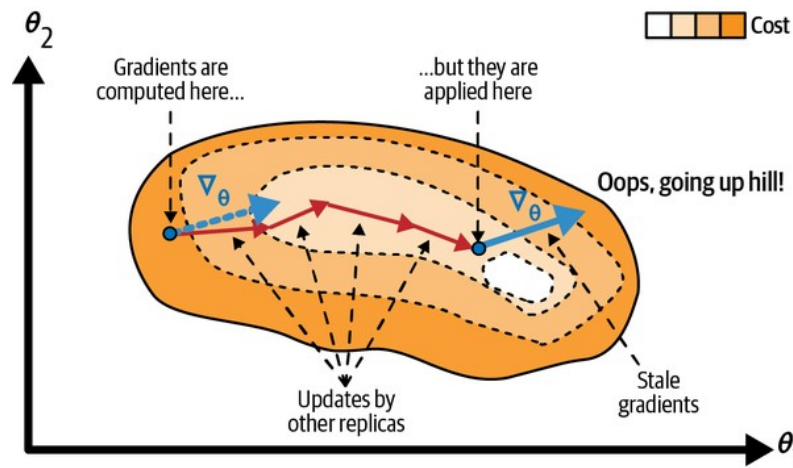


Figure 57: Stale gradients when using asynchronous updates

There are a few ways you can reduce the effect of stale gradients:

- Reduce the learning rate.
- Drop stale gradients or scale them down.
- Adjust the mini-batch size.
- Start the first few epochs using just one replica (this is called the **warm-up phase**). Stale gradients tend to be more damaging at the beginning of training, when gradients are typically large and the parameters have not settled into a valley of the cost function yet, so different replicas may push the parameters in quite different directions.

From benchmarks against previous work on various approaches, it was found that using synchronous updates with a few spare replicas was more efficient than using asynchronous updates, not only converging faster but also producing a better model.

### Bandwidth saturation

Whether synchronous or asynchronous updates are used, data parallelism with centralized parameters still requires communicating the model parameters from the parameter servers to every replica at the beginning of each training step, and the gradients in the other direction at the end of each training step. Similarly, when using the mirrored strategy, the gradients produced by each GPU will need to be shared with every other GPU. Unfortunately, there often comes a point where **adding an extra GPU will not improve performance** at all because the time spent moving the data into and out of GPU RAM (and across the network in a distributed setup) will outweigh the speedup obtained by splitting the computation load. At that point, adding more GPUs will just worsen the bandwidth saturation and actually slow down training. Saturation is more severe for large dense models, since they have a lot of parameters and gradients to transfer. It is less severe for small models (but the parallelization gain is limited) and for large sparse models, where the gradients are typically mostly zeros and so can be communicated efficiently.

## Training at Scale Using the Distribution Strategies API

TensorFlow comes with a very nice API that takes care of all the complexity of distributing your model across multiple devices and machines: the **distribution strategies API**. To train a Keras model across all available GPUs (on a single machine) using data parallelism with the mirrored strategy, just a **MirroredStrategy** object is created, its **scope()** method is called to get a **distribution context**, and the creation and compilation of the model are wrapped inside that context. Then the model's **fit()** method is called normally:

```
strategy = tf.distribute.MirroredStrategy()

with strategy.scope():
    model = tf.keras.Sequential([...])    # create a Keras model normally
    model.compile([...])                  # compile the model normally

batch_size = 100                        # preferably divisible by the number of replicas
model.fit(X_train, y_train, epochs=10,
          validation_data=(X_valid, y_valid), batch_size=batch_size)
```

Under the hood, Keras is distribution-aware, so in this **MirroredStrategy** context it knows that it must replicate all variables and operations across all available GPU devices and hence its model's weights are of type **MirroredVariable**.

The **fit()** method will **automatically split each training batch** across all the replicas, so it's preferable to ensure that the batch size is divisible by the number of replicas (i.e., the number of available GPUs) so that all replicas get batches of the same size. Training will generally be significantly faster than using a single device, and the code change was really minimal.

To make predictions efficiently, the **predict()** method is called, and it will automatically split the batch across all replicas, making predictions in parallel. Again, the batch size must be divisible by the number of replicas. If the model's **save()** method is called, it will be saved as a regular model, not as a mirrored model with multiple replicas. So when it gets loaded, it will run like a regular model, on a single device: by default on GPU #0, or on the CPU if there are no GPUs. If it is required for model to be loaded on all available devices, you **tf.keras.models.load\_model()** within a distribution context is called.

```
with strategy.scope():
    model = tf.keras.models.load_model("my_mirrored_model")
```

If only a subset of all the available GPU devices should be used, the list can be passed into the **MirroredStrategy**'s constructor.

```
strategy = tf.distribute.MirroredStrategy(devices=["/gpu:0", "/gpu:1"])
```

For data parallelism with centralized parameters, the **CentralStorageStrategy** instead of **MirroredStrategy** should be used.

```
strategy = tf.distribute.experimental.CentralStorageStrategy()
```

Optionally, the list of devices to be used as workers can be specified by setting `compute_devices` argument—by default it will use all available GPUs—and optionally the `parameter_device` argument can be set to specify the device to store the parameters on. By default it will use the CPU, or the GPU if there is just one.

### ***Training a Model on a TensorFlow Cluster***

A *TensorFlow cluster* is a group of TensorFlow processes running in parallel, usually on different machines, and talking to each other to complete some work—for example, training or executing a neural network model. Each TF process in the cluster is called **a task**, or **a TF server**. It has an IP address, a port, and a type (also called its **role** or its **job**). The type can be either **"worker"**, **"chief"**, **"ps"** (parameter server), or **"evaluator"**:

- Each **worker performs computations**, usually on a machine with one or more GPUs.
- The **chief performs computations** as well (it is a worker), but it also **handles extra work** such as writing TensorBoard logs or saving checkpoints. There is a **single chief** in a cluster. If no chief is specified explicitly, then by convention the first worker is the chief.
- A **parameter server only keeps track of variable values**, and it is usually on a CPU-only machine. This type of task is only used with the `ParameterServerStrategy`.
- An evaluator obviously takes care of evaluation. This type is **not used often**, and when it's used, there's usually just one evaluator.

To start a TensorFlow cluster, its **specification is first defined**. This means defining each task's **IP address, TCP port, and type**. For example, the following cluster specification defines a cluster with three tasks (two workers and one parameter server; see figure below). The cluster spec is a dictionary with one key per job, and the values are lists of task addresses (IP:port):

```
cluster_spec = {
    "worker": [
        "machine-a.example.com:2222",    # /job:worker/task:0
        "machine-b.example.com:2222"    # /job:worker/task:1"
    ],
    "ps": ["machine-a.example.com:2221"] # /job:ps/task:0
}
```

In general there will be a single task per machine, but as this example shows, multiple tasks can also be configured on the same machine. In this case, if they share the same GPUs, appropriate RAM split needs to be ensured.

Before starting a task, it must be provided with cluster specification and this can be done by setting `TF_CONFIG` environment variable before starting TensorFlow.

```
os.environ["TF_CONFIG"] = json.dumps({
    "cluster": cluster_spec,
    "task": {"type": "worker", "index": 0}
})
```



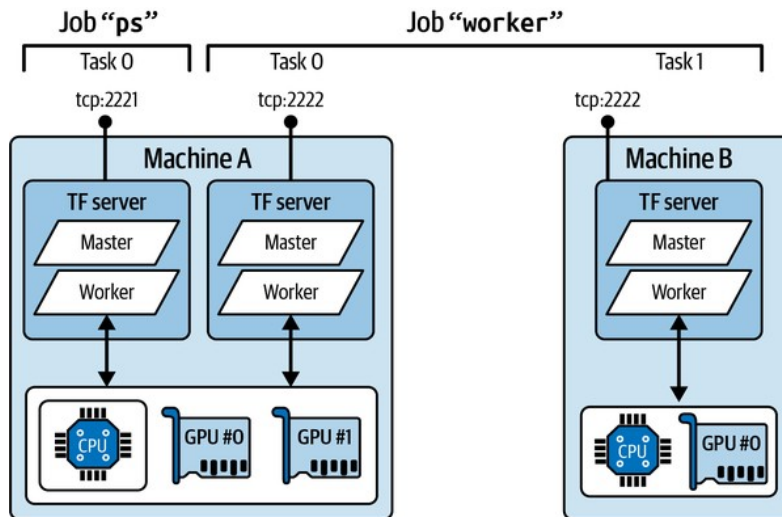


Figure 58: An example TensorFlow cluster

To train a model on a cluster with the mirrored strategy, the `TF_CONFIG` environment variable is set first appropriately for each task. A different task index for each task needs to be ensured. Finally, the following script is run on every worker.

```
import tempfile
import tensorflow as tf

strategy = tf.distribute.MultiWorkerMirroredStrategy()      # at the start
resolver = tf.distribute.cluster_resolver.TFConfigClusterResolver()
print(f"Starting task {resolver.task_type} #{resolver.task_id}")
[...] # load and split the MNIST dataset

with strategy.scope():
    model = tf.keras.Sequential([...]) # build the Keras model
    model.compile(...) # compile the model
    model.fit(X_train, y_train, validation_data=(X_valid, y_valid), epochs=10)

if resolver.task_id == 0: # the chief saves the model to the right location
    model.save("my_mnist_multiworker_model", save_format="tf")
else:
    tmpdir = tempfile.mkdtemp() # other workers save to a temporary directory
    model.save(tmpdir, save_format="tf")
    tf.io.gfile.rmtree(tmpdir) # this directory can be deleted at the end
```

When the script with **MultiWorkerMirroredStrategy** is run on the first workers, they will remain blocked at the AllReduce step, but training will begin as soon as the last worker starts up, and it can be noticed that all are advancing at exactly the same rate since they synchronize at each step.

There are two AllReduce implementations for this distribution strategy: a ring AllReduce algorithm based on gRPC for the network communications, and NCCL's implementation. The best algorithm to use depends on the number of workers, the number and types of GPUs, and the network. By default, TensorFlow will apply some heuristics to select the right algorithm, but a specific AllReduce implementation can be forced like this:



```
strategy = tf.distribute.MultiWorkerMirroredStrategy(  
    communication_options=tf.distribute.experimental.CommunicationOptions(  
        implementation=tf.distribute.experimental.CollectiveCommunication.NCCL))
```

To implement asynchronous data parallelism with parameter servers, the strategy is changed to **ParameterServerStrategy**, one or more parameter servers are added, and `TF_CONFIG` is configured appropriately for each task. Although the workers will work asynchronously, the replicas on each worker will work synchronously.

For TPUs on Google Cloud—for example, on Colab, the accelerator type is set to TPU—then a `TPUStrategy` is created as shown below. This needs to be run right after importing TensorFlow. With this models can be trained across multiple GPUs and multiple servers.

```
resolver = tf.distribute.cluster_resolver.TPUClusterResolver()  
tf.tpu.experimental.initialize_tpu_system(resolver)  
strategy = tf.distribute.experimental.TPUStrategy(resolver)
```

## Exercises

1. What does a `SavedModel` contain? How do you inspect its content?
2. When should you use TensorFlow (TF) Serving? What are its main features? What are some tools you can use to deploy it?
3. How do you deploy a model across multiple TF Serving instances?
4. When should you use the gRPC API rather than the REST API to query a model served by TF Serving?
5. How does TensorFlow achieves parallelism across multiple devices?
6. What are model parallelism and data parallelism? Why is the latter generally recommended?
7. When training a model across multiple servers in TF Cluster, what distribution strategies can you use? How do you choose which one to use with respect to synchronous and asynchronous data parallelism?

# MODULE 4

<PLACEHOLDER>

**MODULE 5**

<PLACEHOLDER>

## REFERENCES

1. Aurelien Geron, *Hands on Machine Learning with Scikit-Learn & TensorFlow*. O'Reilly, 3rd Edition, 2022
2. Ian Goodfellow, Yoshua Bengio and Aaron Courville. *Deep Learning*, MIT Press, 2016
3. Charu C. Aggarwal. *Neural Networks and Deep Learning*, Springer, 2nd Edition, 2023