Neural Networks & Deep Learning [18AI81]

Teaching Material

Version 0.9.4

Instructor: Pradip Kumar Das

Companion source code is available at https://github.com/PradipKumarDas/Teaching/tree/master/18AI81-Neural_Networks_and_Deep_Learning/

Table of Contents

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.

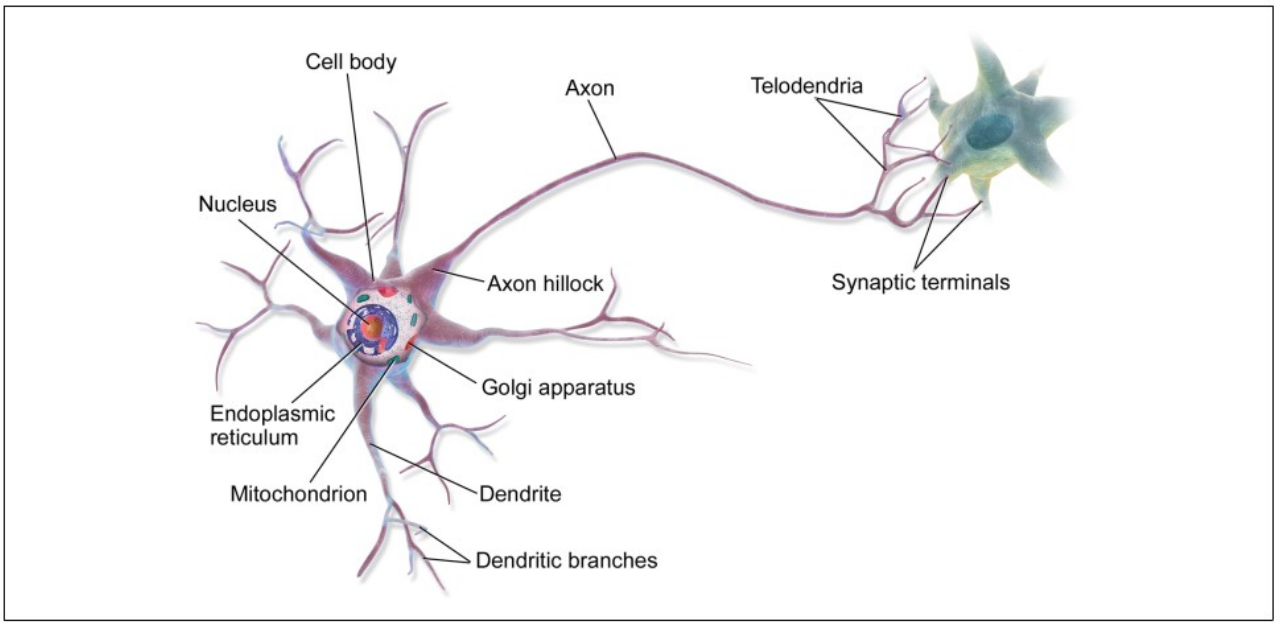


Figure 1: A biological neuron

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 build to perform various operations as shown below.

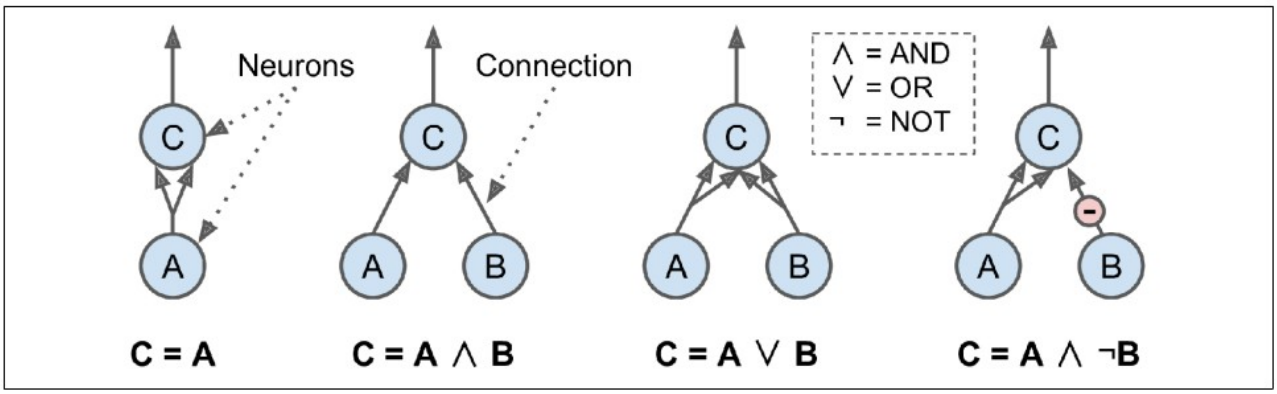


Figure 2: ANNs performing simple logical computations

- 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: $z = w_1 x_1 + w_2 x_2 + \cdots + w_n x_n = \mathbf{w}^T \mathbf{x} + b$. 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.

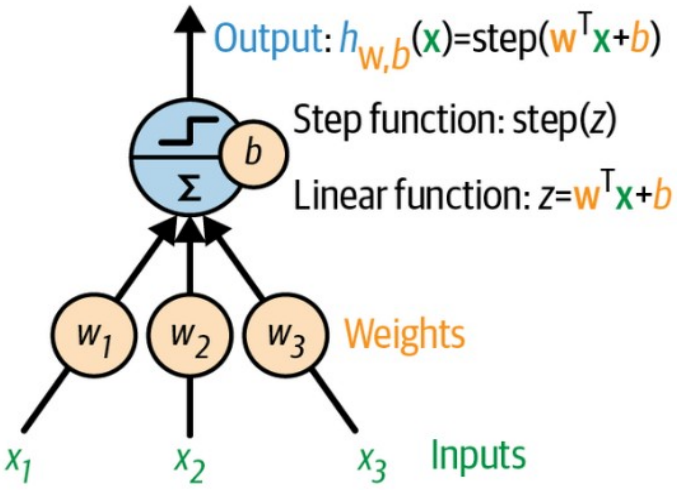


Figure 3: TLU: an artificial neuron that computes a weighted sum of its inputs

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

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

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.

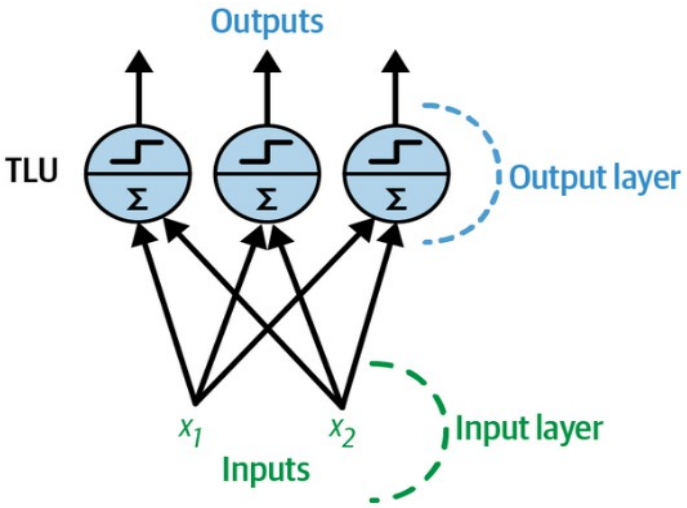


Figure 4: Architecture of a perceptron with two inputs and three output neurons

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.

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

Equation 2: Computing the outputs of a fully connected layer

In this equation:

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

- The bias vector ***b*** contains all the bias terms: one per neuron.
- The function ϕ 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.

$$w_{i,j}^{(next\ step)} = w_{i,j} + \eta\left(y_j - \hat{y}_j\right)x_i$$

Equation 3: Perceptron learning rule (weight update)

In this equation:

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

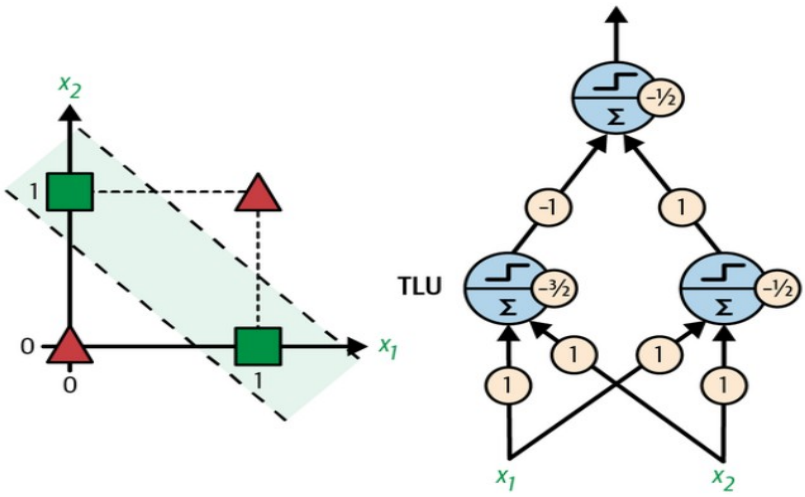


Figure 5: solving XOR function using an MLP

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).

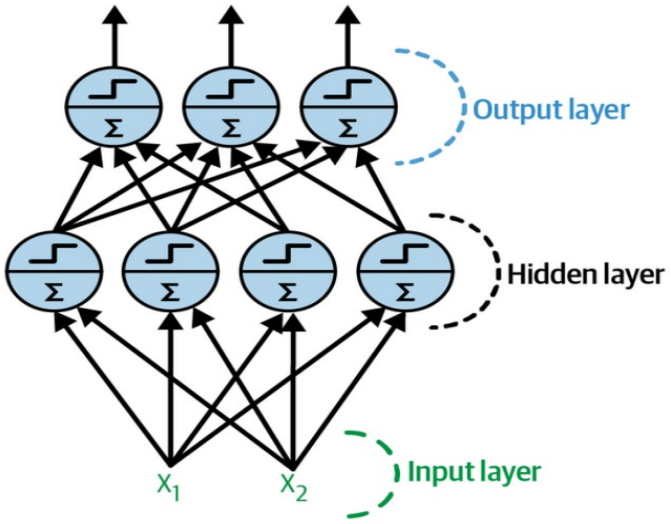


Figure 6: Architecture of a multilayer perceptron with two inputs, one hidden layer of four neurons, and three output neurons

In general, an ANN contains two or more stack of hidden layers, is called a *deep neural network* (DNN). Gradient descent can be used 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:

$$\tanh(z) = 2\sigma(2z) - 1 = \frac{2}{1+\exp(-2z)} - 1$$

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:

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

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.

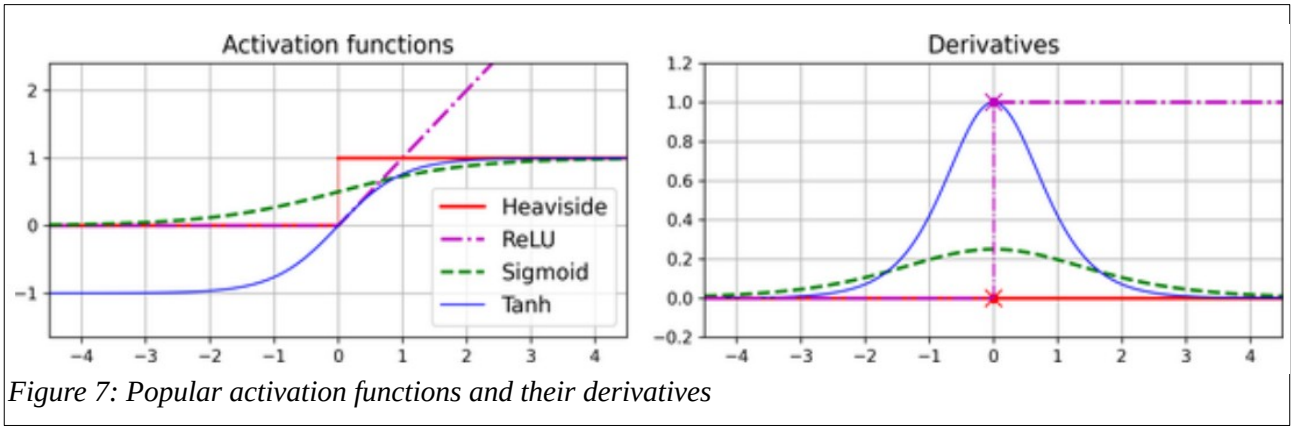


Figure 7: Popular activation functions and their derivatives

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 ℓ_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: $softplus(z) = \log(1 + \exp(z))$. 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.

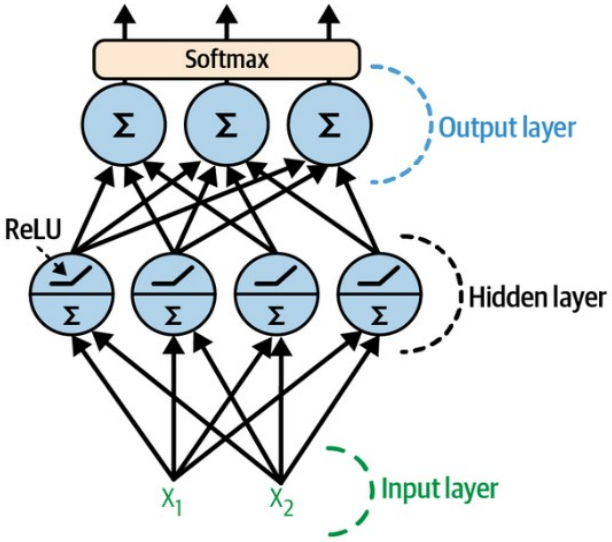


Figure 8: A MLP (with ReLU and softmax) for classification

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 × 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's 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()
```

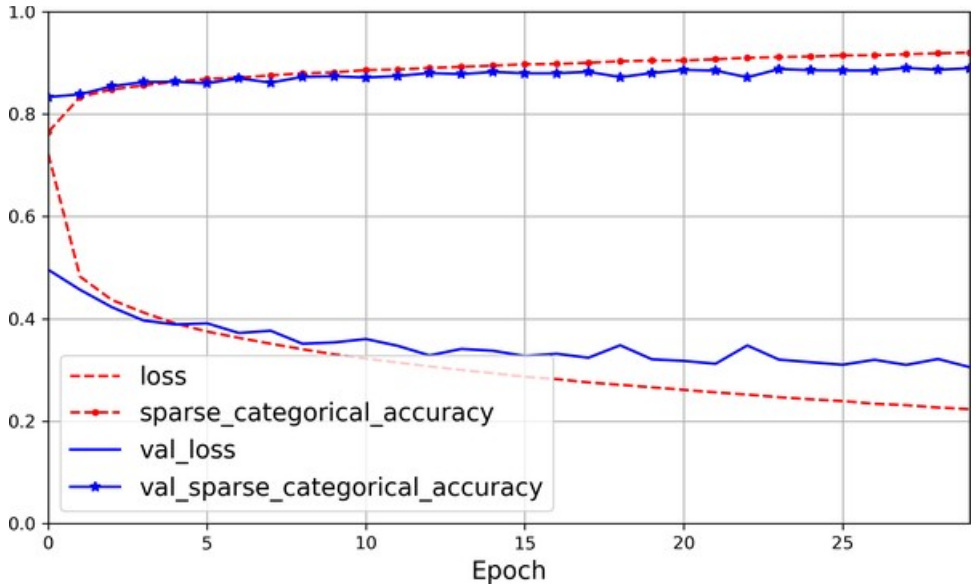


Figure 10: Learning curves: the mean loss and accuracy measured over each epoch, both for training and validation data

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 optimzer 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).

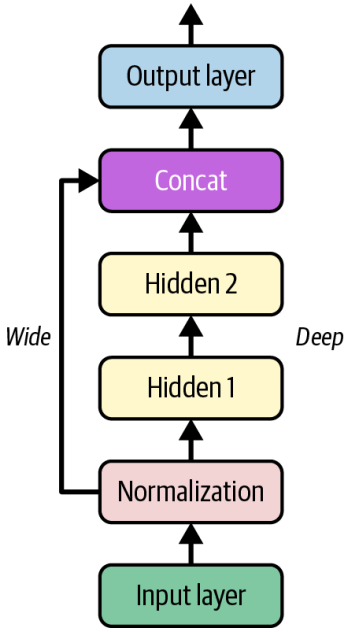


Figure 11: Wide & Deep neural network

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 though the figure and source code below.

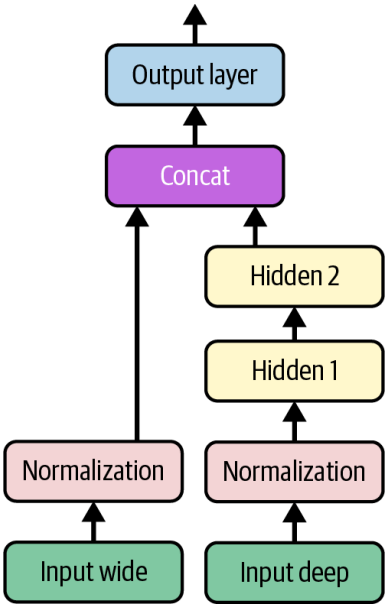


Figure 12: Handling multiple inputs

```
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.

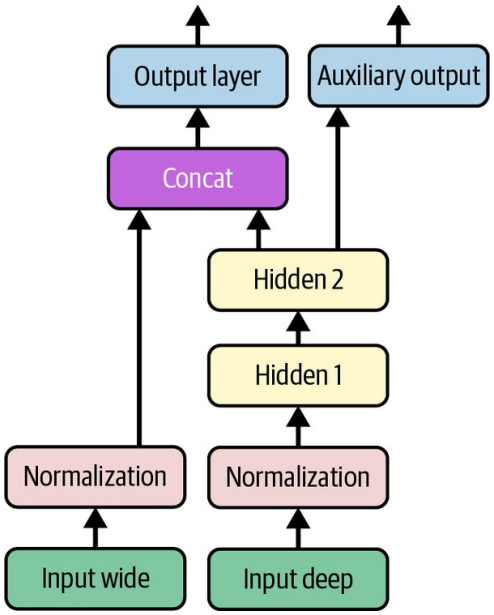


Figure 13: Handling multiple outputs

```
[...] # 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 an 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 bed compile it, 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 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 mode 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 parametere `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

`on_train_begin()`
`on_train_end()`
`on_epoch_begin()`
`on_epoch_end()`
`on_batch_begin()`
`on_batch_end()`
	`on_test_begin()`
	`on_test_end()`
	`on_test_batch_begin()`
	`on_test_batch_end()`
		`on_predict_begin()`
		`on_predict_end()`
		`on_predict_batch_begin()`
		`on_predict_batch_end()`

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.

```
my_logs
├── run_2022_08_01_17_25_59
│   ├── train
│   │   ├── events.out.tfevents.1659331561.my_host_name.42042.0.v2
│   │   ├── events.out.tfevents.1659331562.my_host_name.profile-empty
│   │   └── plugins
│   │       └── profile
│   │           └── 2022_08_01_17_26_02
│   │               ├── my_host_name.input_pipeline.pb
│   │               └── [...]
│   └── validation
│       └── events.out.tfevents.1659331562.my_host_name.42042.1.v2
└── run_2022_08_01_17_31_12
    └── [...]
```

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.

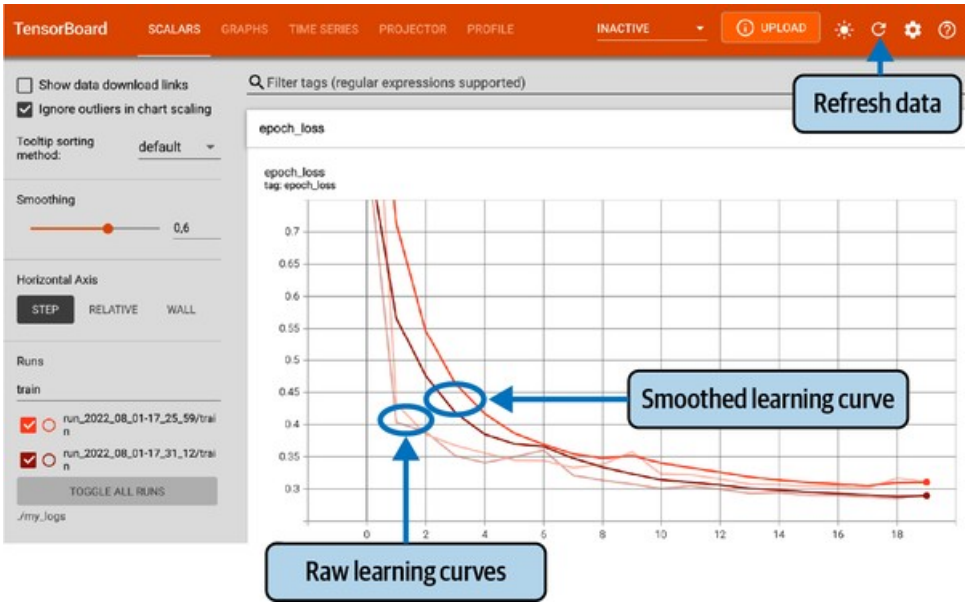


Figure 15: Visualizing learning curves with TensorBoard

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 initializes 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 × 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 **X**?
b) What are the shapes of the hidden layer's weight matrix $\mathbf{W}_h$ and bias vector $\boldsymbol{b}_h$?
c) What are the shapes of the output layer's weight matrix $\mathbf{W}o$ and bias vector $\boldsymbol{b}_o$?
d) What is the shape of the network's output matrix **Y**?
e) Write the equation that computes the network's output matrix **Y** as a function of **X**, $\mathbf{W}_h$, $\boldsymbol{b}_h$, $\mathbf{W}_o$, and $\boldsymbol{b}_o$.

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.

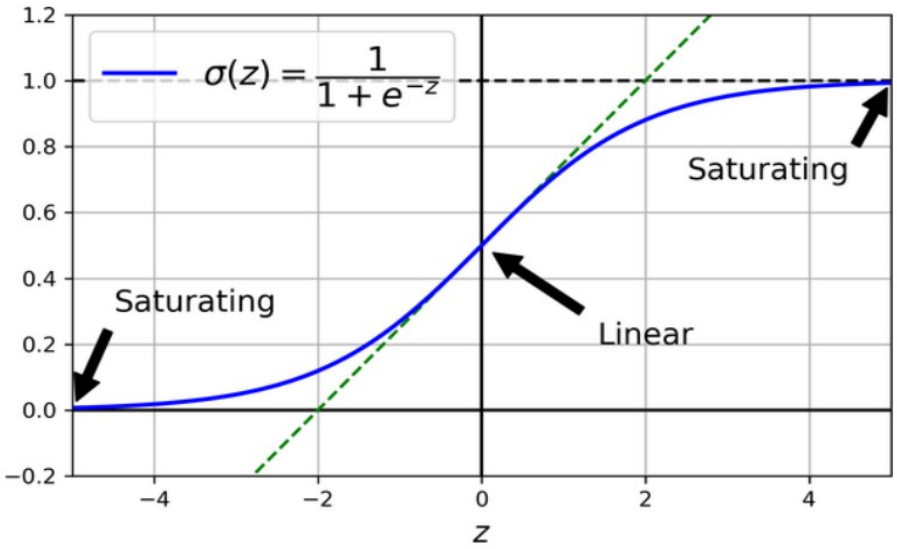


Figure 16: Sigmoid activation function saturation

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*.

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

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

Figure 17: Glorot initialization for sigmoid activation function

LeCun initialization can be derived if fan_{avg} is replace 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	σ^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 α defines how much the function "leaks": it is the slope of the function for $z < 0$ to ensures 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 α = 0.01. Evaluation on *randomized leaky ReLU* (RReLU) where *α* 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 *α* 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.

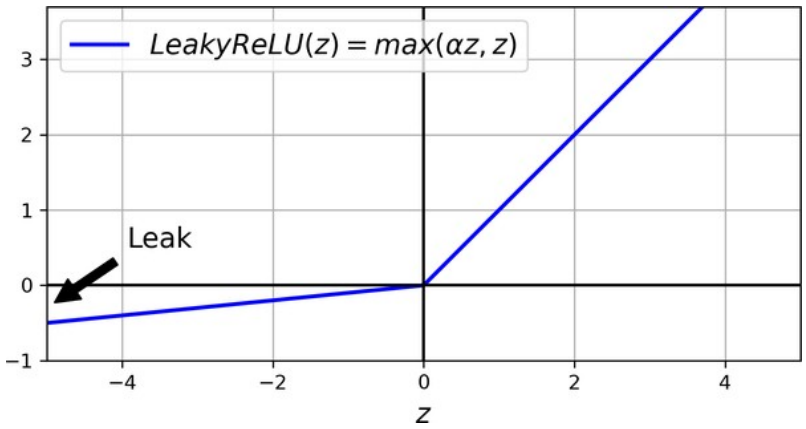


Figure 18: Leaky ReLu: A ReLu with a small slope for negative values

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.

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

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 α 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 α 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 $z = 0$.

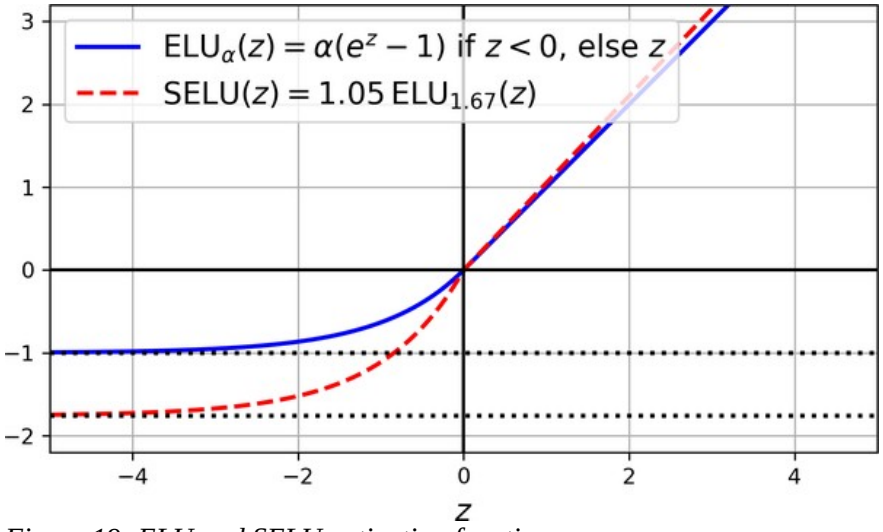


Figure 19: ELU and SELU activation functions

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 α ≈ 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 ℓ_1 or ℓ_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 Φ is the standard Gaussian cumulative distribution function (CDF) and Φ(*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 seem 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 σ 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 *β* 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 $\beta = 1.702$. *β* can be used as a hyperparameter as a trainable parameter much like PReLU and let gradient descent optimize it.

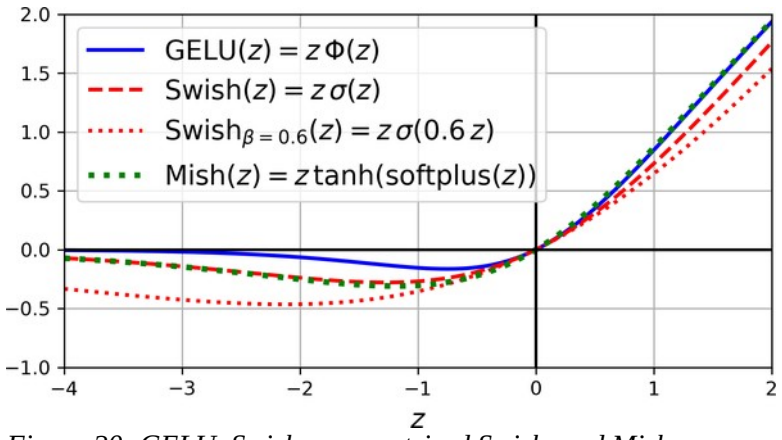


Figure 20: GELU, Swish, parametrized Swish, and Mish activation functions

Mish

Another quite similar activation function is *Mish* and is defined as $mish(z) = z\,tanh(softplus(z))$, where $softplus(z) = log(1 + exp(z))$. 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 and even more complex tasks,*

parametrized Swish with a learnable β 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 implement 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 is summarized step by step in the equation below.

1 $\mu_B = \frac{1}{m_B} \sum_{i=1}^{m_B} \mathbf{x}^{(i)}$

2 $\sigma_B^2 = \frac{1}{m_B} \sum_{i=1}^{m_B} (\mathbf{x}^{(i)} - \mu_B)^2$

3 $\hat{\mathbf{x}}^{(i)} = \frac{\mathbf{x}^{(i)} - \mu_B}{\sqrt{\sigma_B^2 + \varepsilon}}$

4 $\mathbf{z}^{(i)} = \gamma \otimes \hat{\mathbf{x}}^{(i)} + \beta$

Equation 7: Batch normalization algorithm

In this algorithm:

- $\boldsymbol{\mu}_B$ 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.

- $\boldsymbol{\sigma}_B$ 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*.
- ε 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.
- γ 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).
- β 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.
- $\mathbf{z}^{(i)}$ 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: $\boldsymbol{\gamma}$ (the output scale vector) and $\boldsymbol{\beta}$ (the output offset vector) are learned through regular backpropagation, and $\boldsymbol{\mu}$ (the final input mean vector) and $\boldsymbol{\sigma}$ (the final input standard deviation vector) are estimated using an exponential moving average. Note that $\boldsymbol{\mu}$ and $\boldsymbol{\sigma}$ 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 $\boldsymbol{XW} + \boldsymbol{b}$, then the BN layer will compute $\boldsymbol{\gamma} \otimes (\boldsymbol{XW} + \boldsymbol{b} - \boldsymbol{\mu}) / \boldsymbol{\sigma} + \boldsymbol{\beta}$ (ignoring the smoothing term ε in the denominator). If we define $\mathrm{W}' = \boldsymbol{\gamma} \otimes \boldsymbol{W} / \boldsymbol{\sigma}$ and $\boldsymbol{b}' = \boldsymbol{\gamma} \otimes (\boldsymbol{b} - \boldsymbol{\mu}) / \boldsymbol{\sigma} + \boldsymbol{\beta}$, the equation simplifies to $\boldsymbol{XW}' + \boldsymbol{b}'$.

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 (BatchN  (None, 784)              3136
 ormalization)

 dense (Dense)               (None, 300)               235500

 batch_normalization_1 (Batc  (None, 300)              1200
 hNormalization)

 dense_1 (Dense)             (None, 100)               30100

 batch_normalization_2 (Batc  (None, 100)              400
 hNormalization)

 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: $\boldsymbol{\gamma}$, $\boldsymbol{\beta}$, $\boldsymbol{\mu}$, and $\boldsymbol{\sigma}$ (for example, the first BN layer adds 3,136 parameters, which is 4 × 784). Parameters, $\boldsymbol{\mu}$ and $\boldsymbol{\sigma}$ are the moving averages and are non-trainable where parameters $\boldsymbol{\gamma}$ and $\boldsymbol{\beta}$ 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.

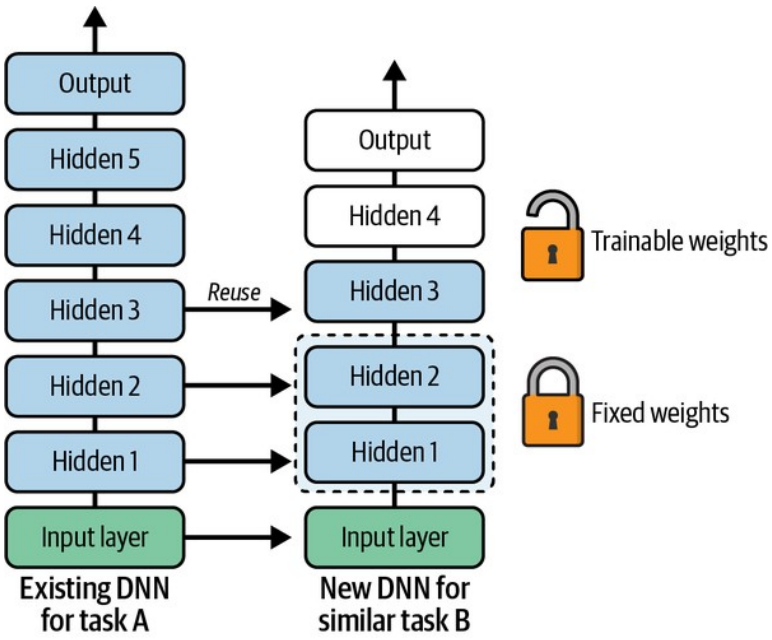
```
optimizer = tf.keras.optimizers.SGD(clipvalue=1.0)
model.compile([...], optimizer=optimizer)
```

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 ℓ_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 loosing 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 unfreezed 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 $\boldsymbol{\theta}$ by directly subtracting the gradient of the cost function $J(\boldsymbol{\theta})$ with regard to the weights ($\nabla_\theta J(\boldsymbol{\theta})$) multiplied by the learning rate η and the equation is $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta\nabla_\theta J(\boldsymbol{\theta})$. 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 $\boldsymbol{m}$ (multiplied by the learning rate η), and it updates the weights by adding this momentum vector (refer equation below).

$$\mathbf{m} \leftarrow \beta \mathbf{m} - \eta \nabla_\theta J(\theta)$$
$$\theta \leftarrow \theta + \mathbf{m}$$

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 β, 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 η 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 $\boldsymbol{\theta}$ but slightly ahead in the direction of the momentum, at $\boldsymbol{\theta} + \beta\mathbf{m}$ (refer equation below).

$$\mathbf{m} \leftarrow \beta \mathbf{m} - \eta \nabla_{\theta} J(\boldsymbol{\theta} + \beta \mathbf{m})$$
$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \mathbf{m}$$

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 ∇_1 represents the gradient of the cost function measured at the starting point θ, and ∇2 represents the gradient at the point located at $\boldsymbol{\theta} + \beta\mathbf{m}$.

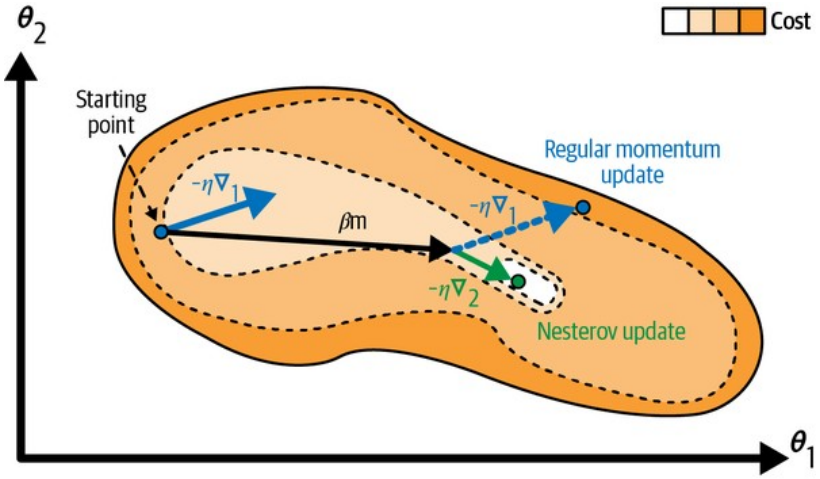


Figure 22: Regular versus Nesterov momentum optimization: the former applies the gradients

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).

$$\mathbf{s} \leftarrow \mathbf{s} + \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) \otimes \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$$
$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) \oslash \sqrt{\mathbf{s} + \varepsilon}$$

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 $\sqrt{\mathbf{s} + \varepsilon}$ (where ε 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 η.

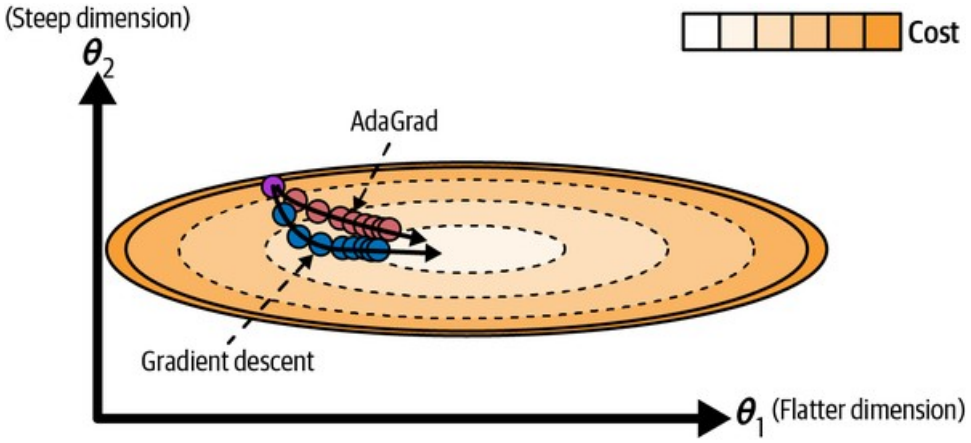


Figure 23: AdaGrad versus gradient descent: the former can correct its direction earlier to point to the optimum

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).

$$\mathbf{s} \leftarrow \rho \mathbf{s} + (1 - \rho) \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) \otimes \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$$
$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) \oslash \sqrt{\mathbf{s} + \varepsilon}$$

Equation 11: RMSProp algorithm

The decay rate ρ (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 $m \leftarrow \beta_1 m - (1 - \beta_1) \nabla_\theta J(\theta)$

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

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

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

5 $\theta \leftarrow \theta + \eta m \oslash \sqrt{s + \varepsilon}$

Adam's close similarity to both momentum optimization and RMSProp can be observed in equation 1, 2 and 5: β_1 is momentum decay hyperparameter (typically initialized to 0.9) and corresponds to β in momentum optimization, and β_2 is scaling decay hyperparameter (often initialized to 0.999) and corresponds to ρ 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 η.

```
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 ε), 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 ℓ_2 norm of the time-decayed gradients and replaces the ℓ_2 norm with the ℓ_∞ 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 ℓ_2 regularization and can be shown mathematically that ℓ_2 regularization is equivalent to weight decay when used on SGD. However, when using Adam or its variants, ℓ_2 regularization and weight decay are *not* equivalent and in practice, combining Adam with ℓ_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 ℓ_1 and ℓ_2 regularization, dropout, and max-norm regularization.

ℓ_1 and ℓ_2 Regularization

ℓ_2 regularization can be used to constrain a neural network's connection weights where ℓ_1 or both ℓ_1 and ℓ_2 regularization can be used to build a sparse model (with many weights equal to 0). The following code shows that ℓ_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 ℓ_1 and ℓ_2 regularization can be applied by using `tf.keras.regularizers.l1_l2()` by specifying both regularization factors.

ℓ_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.

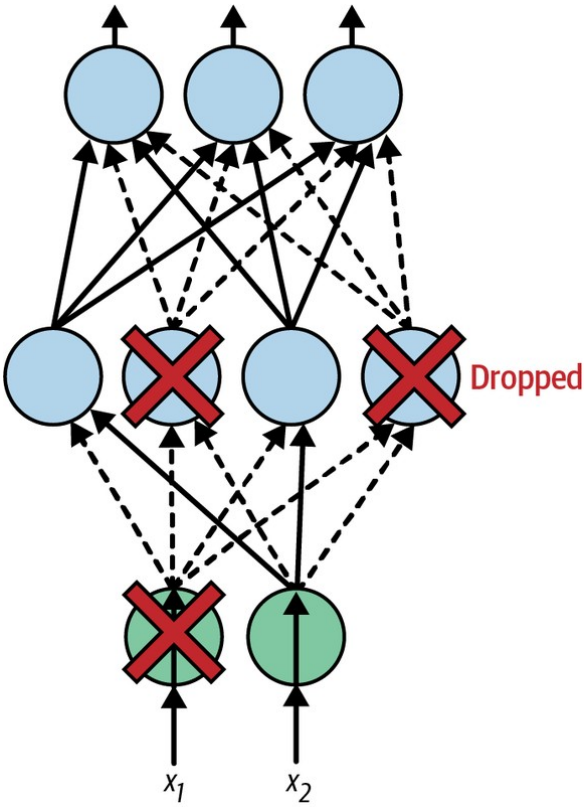


Figure 24: Dropout regularization: at each training iteration, a random subset of all neurons in one or more layers, except the output layer, are dropped out; these neurons output 0 at this iteration (represented by the dashed arrows)

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 – 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")
])
[...] # compile and train the model
```

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 fist 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_probas.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 ℓ_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 optimzer 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.

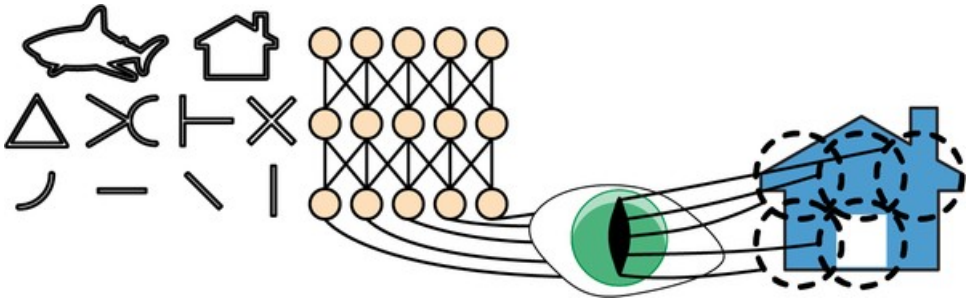


Figure 25: Biological neurons in the visual cortex respond to specific patterns in small regions of the visual field called receptive fields; as the visual signal makes its way through consecutive brain modules, neurons respond to more complex patterns in larger receptive fields

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 × 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 flatten to 1D before feeding to the neural network.

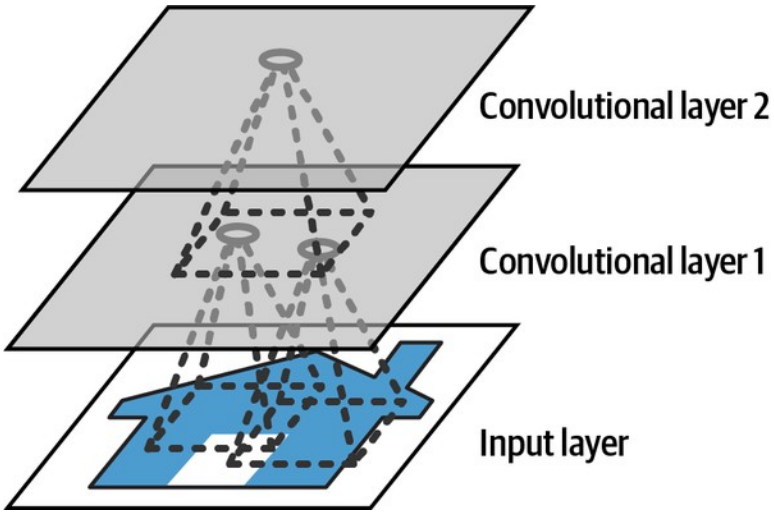


Figure 26: CNN layers with rectangular local receptive fields

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*.

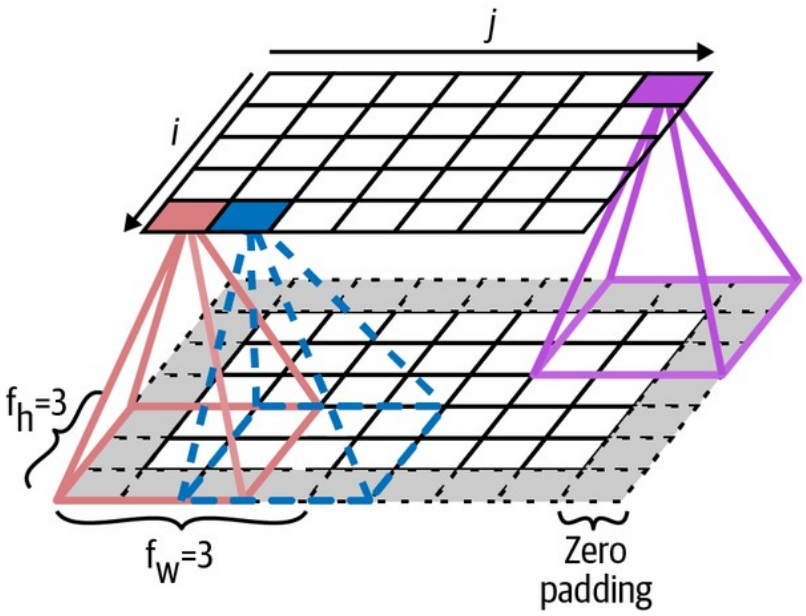


Figure 27: Connections between layers and zero padding

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 × 7 input layer (plus zero padding) is connected to a 3 × 4 layer, using 3 × 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.

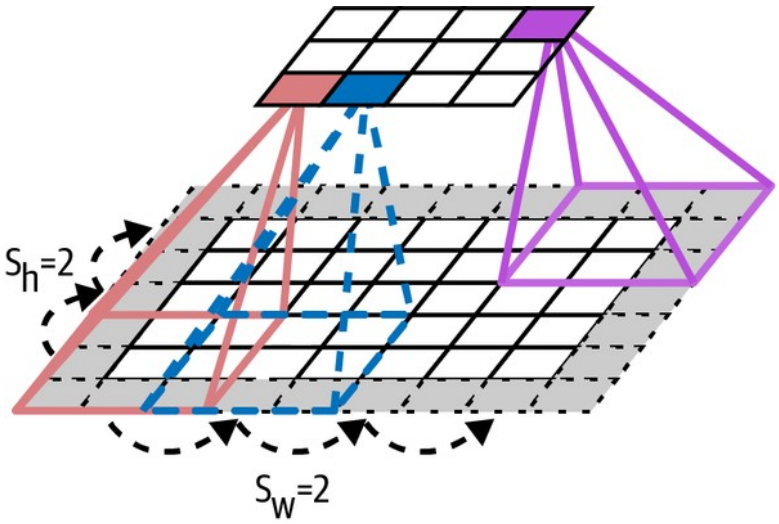


Figure 28: Reducing dimensionality using a stride of 2

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 x 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.

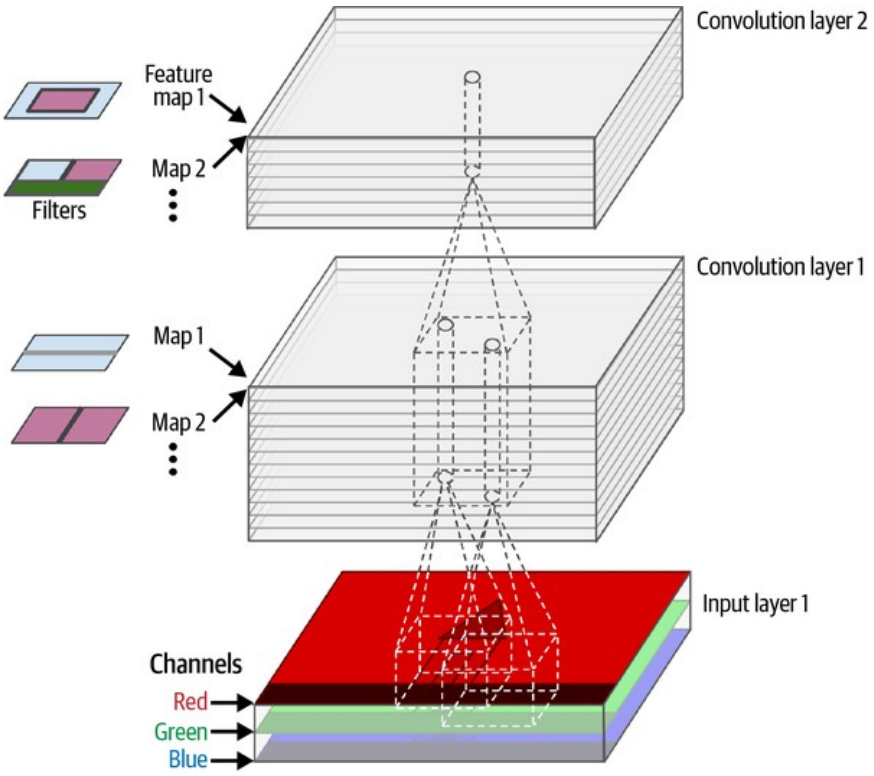


Figure 29: Two convolutional layers with multiple filters each (kernels), processing a color image with three color channels; each convolutional layer outputs one feature map per filter

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.

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

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 × 5 filters, with stride 1 and "same" padding. If the input is a 150 × 100 RGB image (three channels), then the number of parameters is (5 × 5 × 3 + 1) × 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 × 100 neurons, and each of these neurons needs to compute a weighted sum of its 5 × 5 × 3 = 75 inputs: that's a total of 225 million (75 × 150 × 100 × 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 × 150 × 100 × 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 × 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.

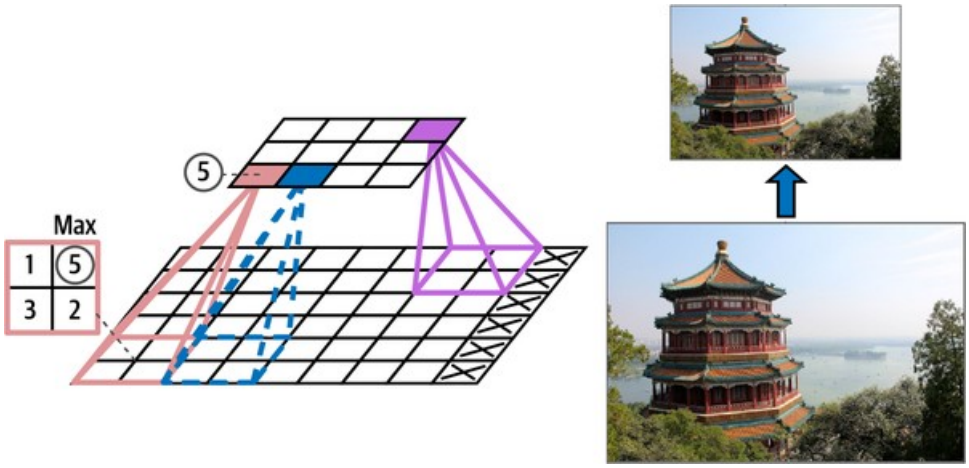


Figure 30: Max pooling layer (2 × 2 pooling kernel, stride 2, no padding)

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 × 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.

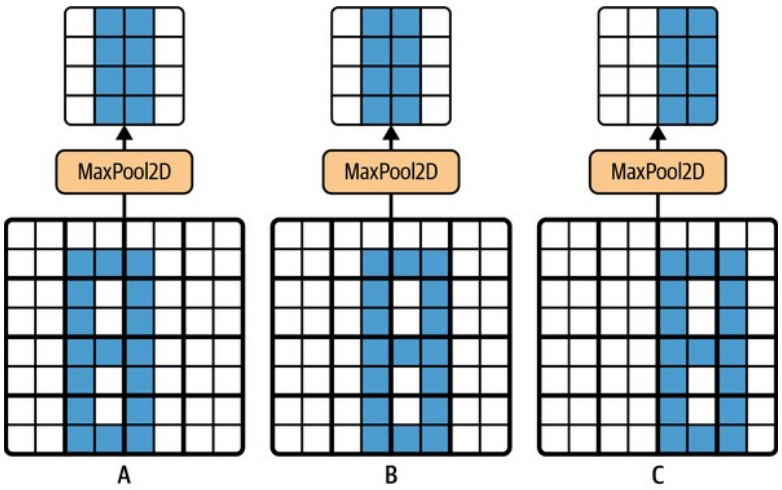


Figure 31: Invariance to small translations

However, max pooling has some downsides too. It's obviously very destructive: even with a tiny 2 × 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 × 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 MaxPool2D. 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.

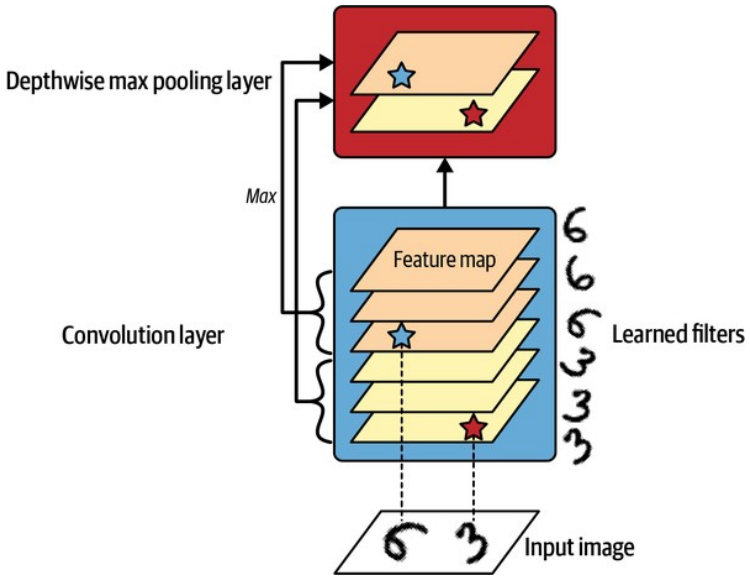


Figure 32: Depthwise max pooling can help the CNN learn to be invariant (to rotation in this case)

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 (+ReLUs), and the final layer outputs the prediction (e.g., a softmax layer that outputs estimated class probabilities).

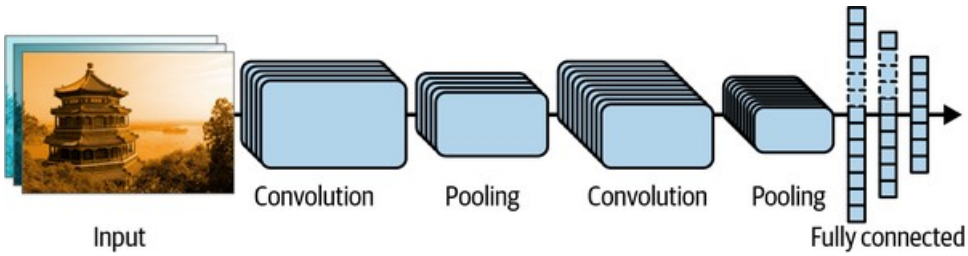


Figure 33: Typical CNN architecture

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 × 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 × 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 × 6	3 × 3	2	valid	–
C7	Convolution	256	13 × 13	3 × 3	1	same	ReLU
C6	Convolution	384	13 × 13	3 × 3	1	same	ReLU
C5	Convolution	384	13 × 13	3 × 3	1	same	ReLU
S4	Max pooling	256	13 × 13	3 × 3	2	valid	–
C3	Convolution	256	27 × 27	5 × 5	1	same	ReLU
S2	Max pooling	96	27 × 27	3 × 3	2	valid	–
C1	Convolution	96	55 × 55	11 × 11	4	valid	ReLU
In	Input	3 (RGB)	227 × 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 × 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 × 1, 3 × 3] and [1 × 1, 5 × 5]) acts like a single

powerful convolutional layer, capable of capturing more complex patterns.

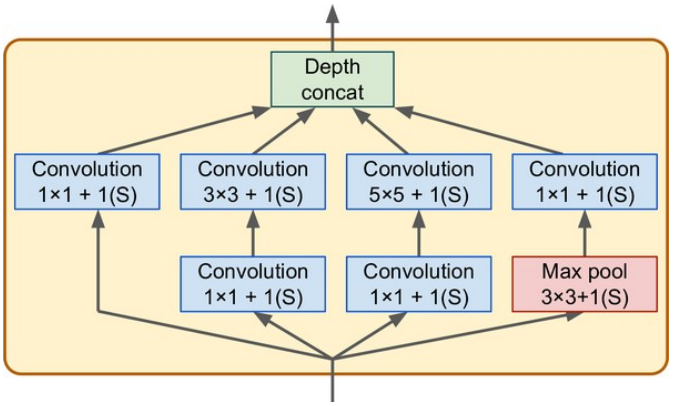


Figure 36: Inception module

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.

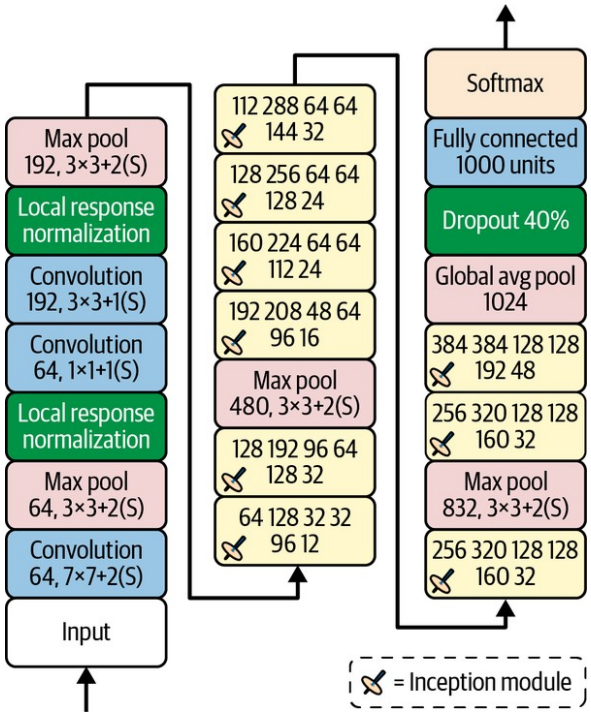


Figure 37: GoogLeNet architecture

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 × 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 × 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).

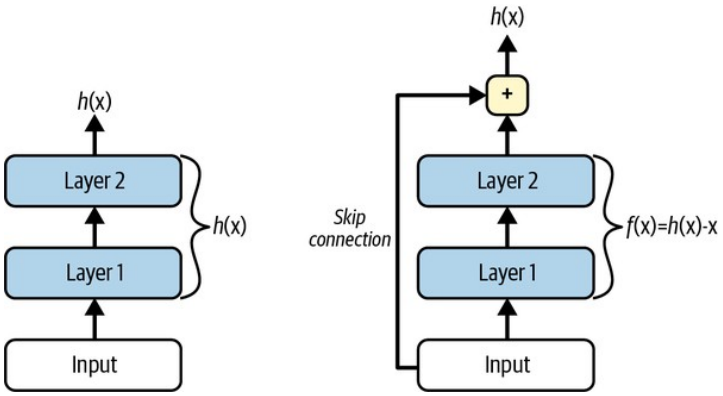


Figure 38: Residual learning

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.

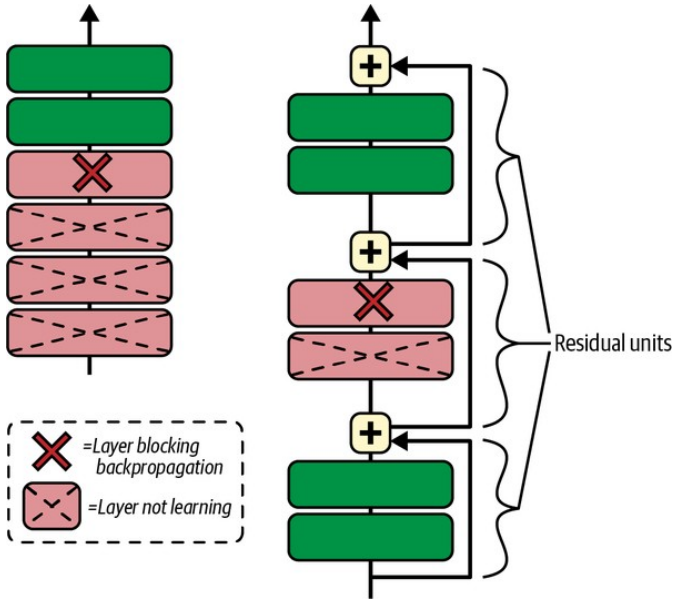


Figure 39: Regular deep neural network (left) and deep residual network (right)

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 × 3 kernels and preserving spatial dimensions (stride 1, "same" padding).

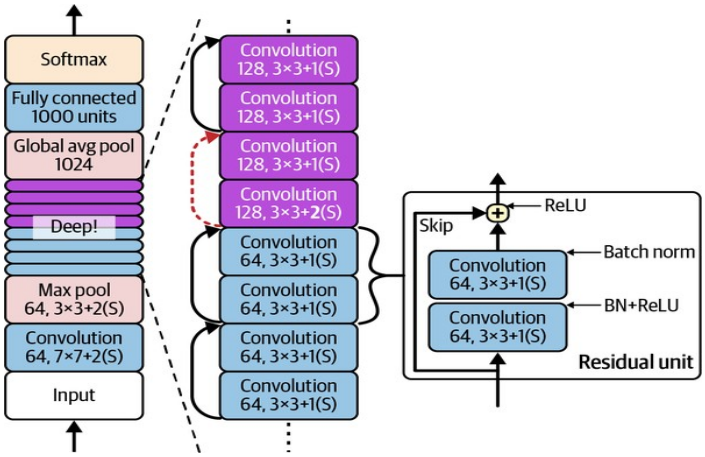


Figure 40: ResNet architecture

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 × 1 convolutional layer with stride 2 and the right number of output feature maps (see next figure).

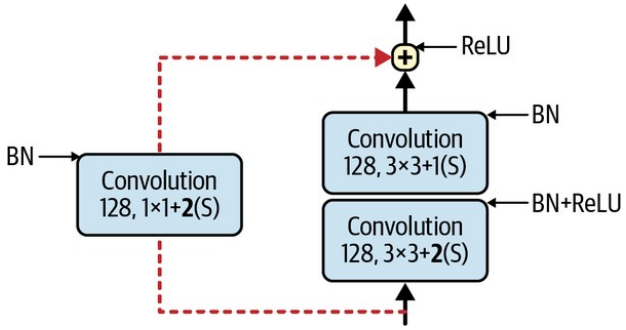


Figure 41: Skip connection when changing feature map size and depth

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 × 1 filters.

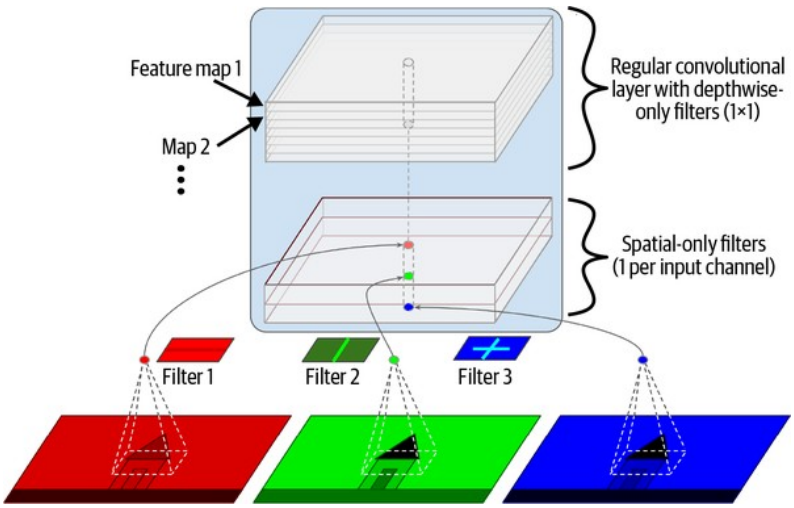


Figure 42: Depthwise separable convolutional layer

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.

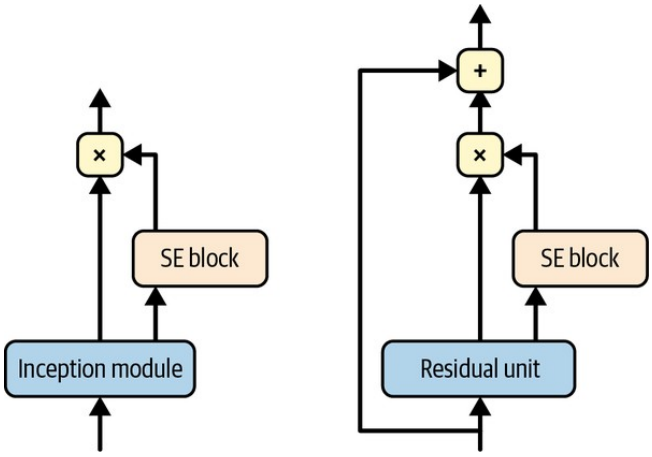


Figure 43: SE-Inception module (left) and SE-ResNet unit (right)

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).

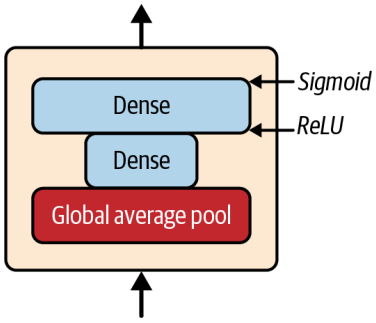


Figure 44: SE block architecture

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 × 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 × 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 with makes reasonable predictions, it might need to be put it 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 done is to export this model to the SavedModel format.

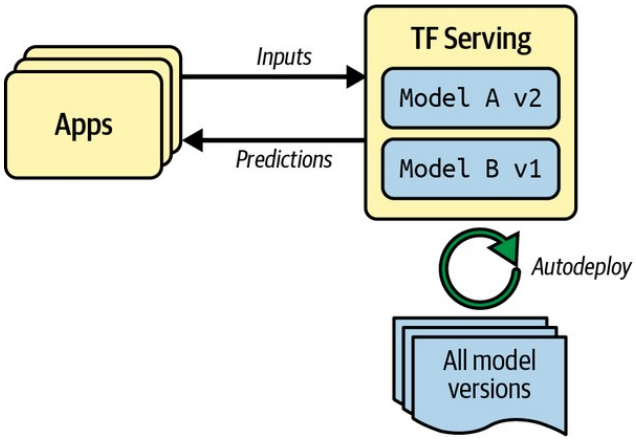


Figure 45: TF Serving can serve multiple models and automatically deploy the latest version of each model

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 reprocessing 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 note 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.

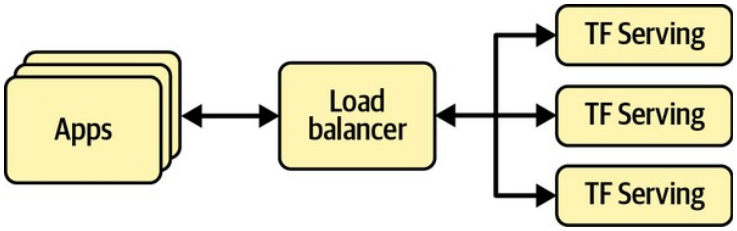


Figure 46: Scaling up TF Serving with load balancing

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.

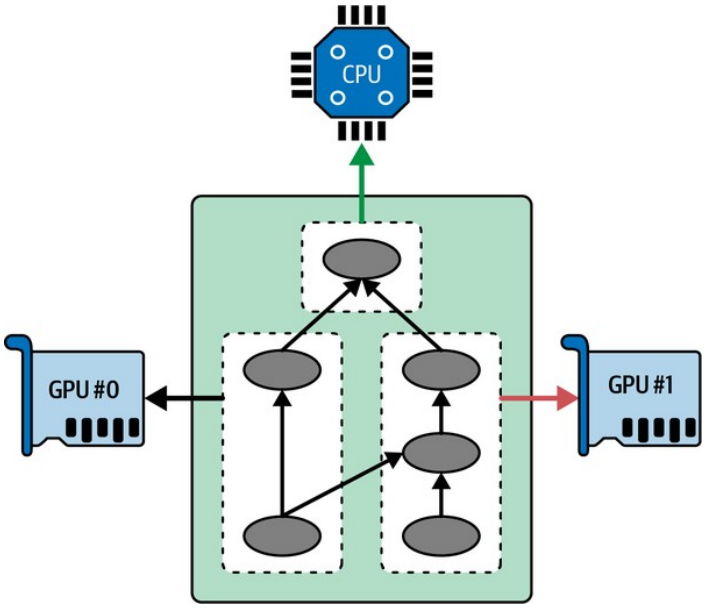


Figure 47: Executing a TensorFlow graph across multiple devices in parallel

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).

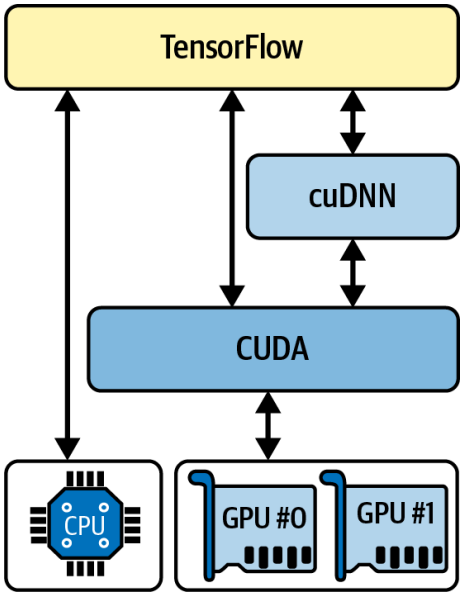


Figure 48: TensorFlow uses CUDA and cuDNN to control GPUs and boost DNNs

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.

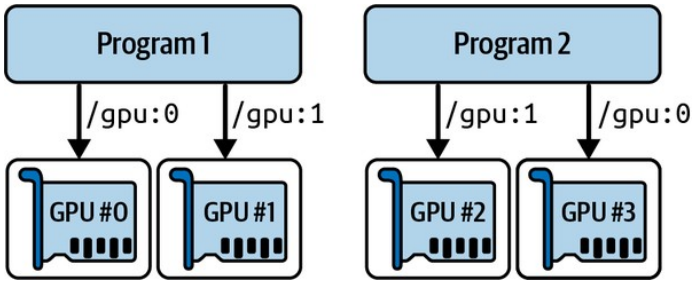


Figure 49: Each program gets two GPUs

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.

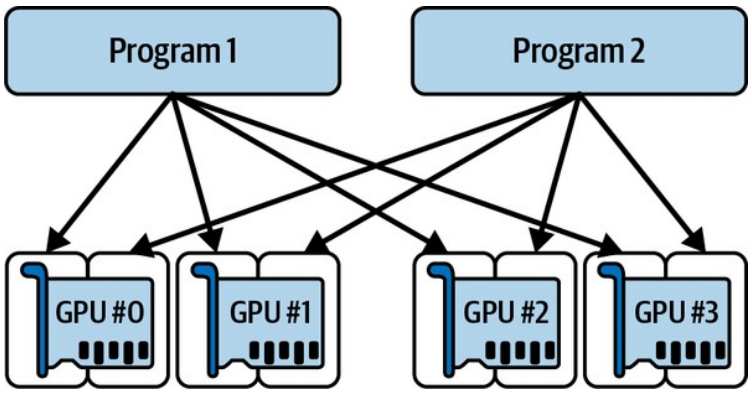


Figure 50: Each program gets all four GPUs, but with only 2 GiB of RAM on each GPU

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.

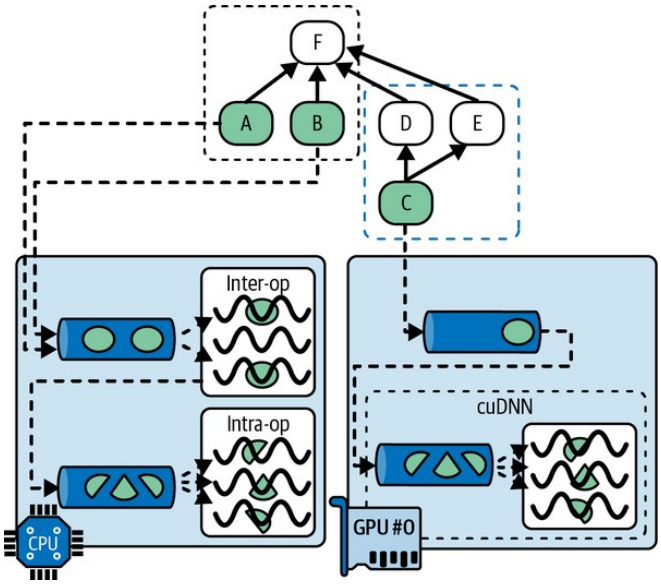


Figure 51: Parallelized execution of a TensorFlow graph

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).

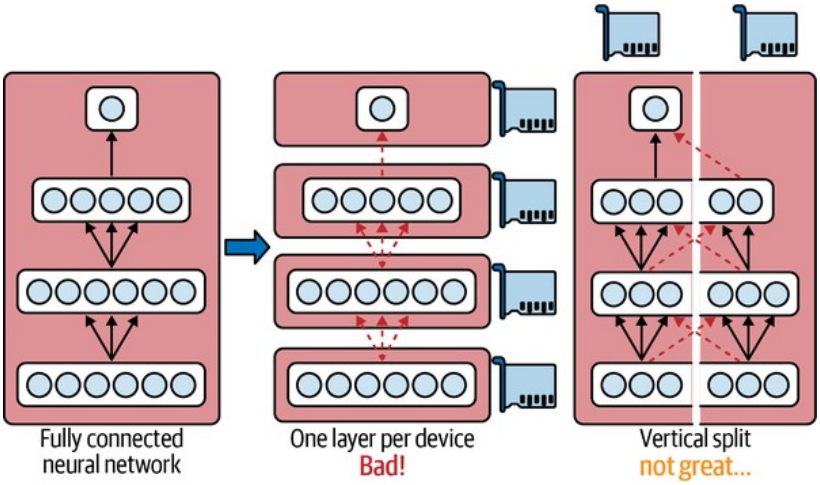


Figure 52: Splitting a fully connected neural network

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).

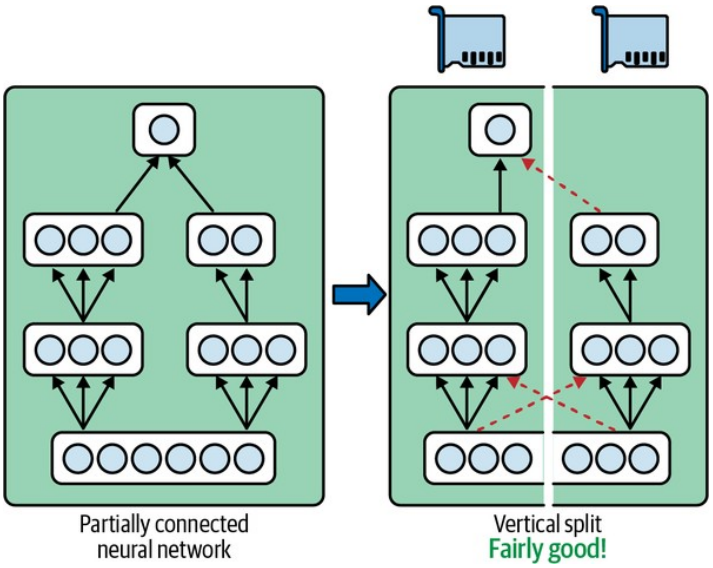


Figure 53: Splitting a partially connected neural network

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.

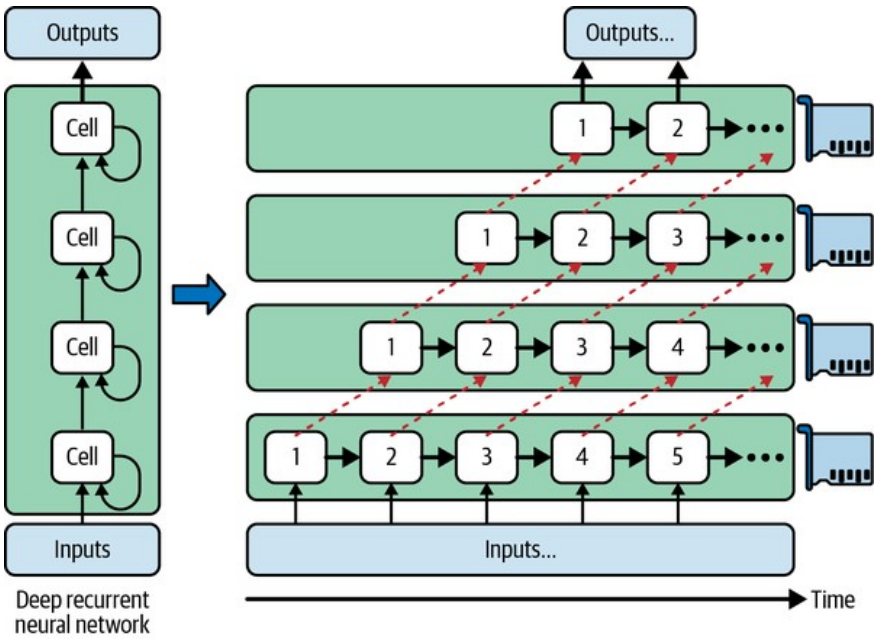


Figure 54: Splitting a deep recurrent neural network

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).

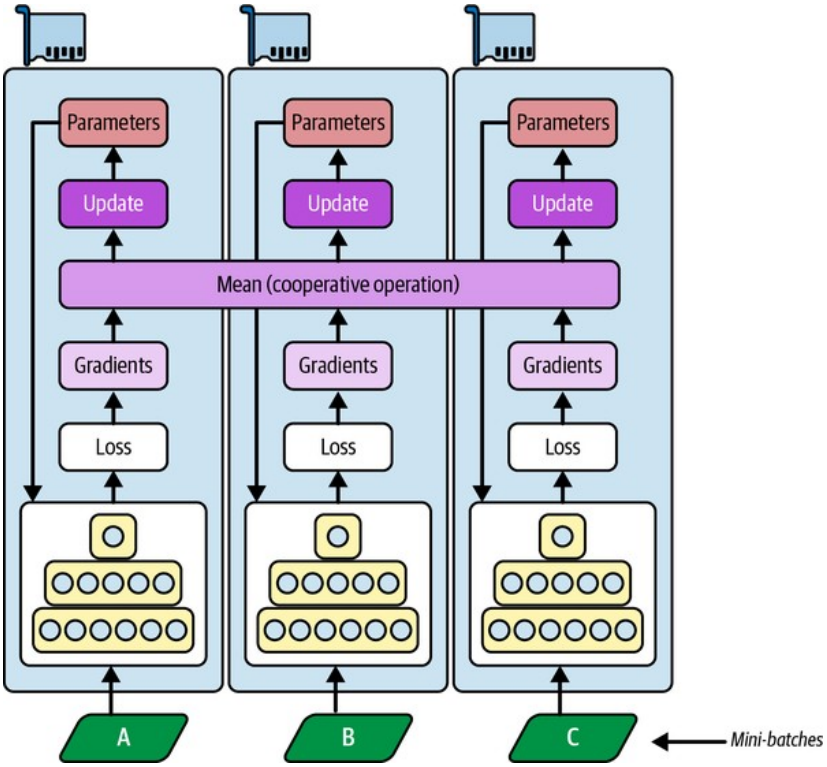


Figure 55: Data parallelism using the mirrored strategy

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.

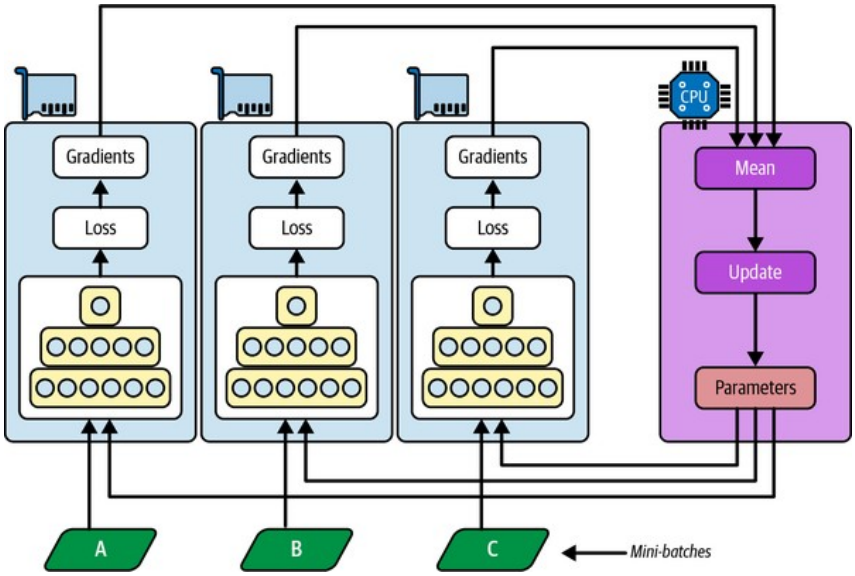


Figure 56: Data parallelism with centralized parameters

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.

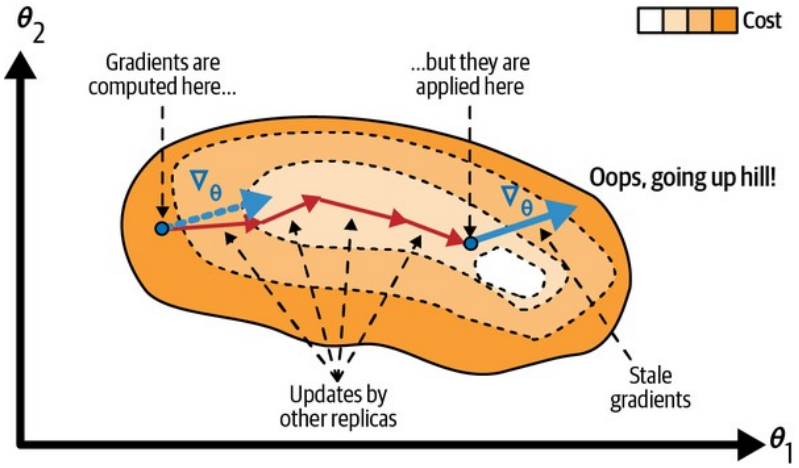


Figure 57: Stale gradients when using asynchronous updates

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 staring 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}
})
```

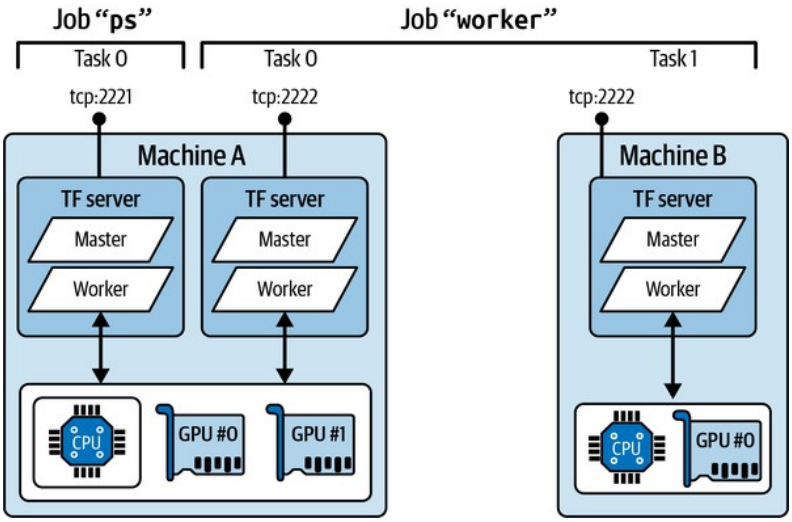


Figure 58: An example TensorFlow cluster

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

Recurrent Neural Network

Recurrent neural networks (RNNs)—a class of nets that can predict the future (well, up to a point). RNNs can analyze time series data, such as the number of daily active users on a website, the hourly temperature in the city, home's daily power consumption, the trajectories of nearby cars, and more. Once an RNN learns past patterns in the data, it is able to use its knowledge to forecast the future, assuming of course that past patterns still hold in the future.

More generally, RNNs can work on sequences of arbitrary lengths, rather than on fixed-sized inputs. For example, they can take sentences, documents, or audio samples as input, making them extremely useful for natural language processing applications such as automatic translation or speech-to-text.

Recurrent Neurons

A recurrent neural network looks very much like a feedforward neural network where the activations flow only in one direction, from the input layer to the output layer, but it also has connections pointing backward.

Below one is a simplest possible RNN, composed of one neuron receiving inputs, producing an output (scaler), and sending that output back to itself, as shown in below figure (left). At each time step *t* (also called a frame), this recurrent neuron receives the inputs $x_{(t)}$ as well as its own output from the previous time step, $\hat{y}_{(t-1)}$. Since there is no previous output at the first time step, it is generally set to 0. This tiny network can be represented against the time axis, as shown in below figure (right). This is called unrolling the network through time (it's the same recurrent neuron represented once per time step).

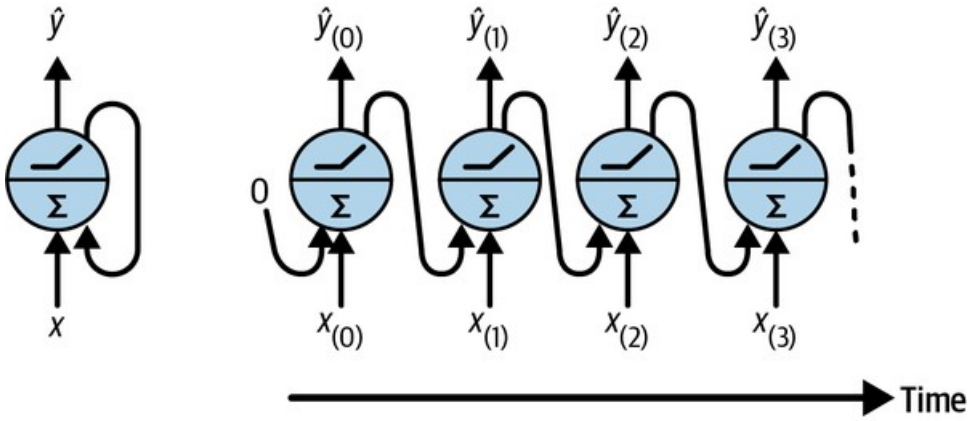


Figure 59: A recurrent neuron (left) unrolled through time (right)

A layer of recurrent neurons can also be created easily. At each time step *t*, every neuron receives both the input vector $x_{(t)}$ and the output vector from the previous time step $\hat{y}_{(t-1)}$, as shown in below figure. It is to be noted that both the inputs and outputs are now vectors (when there was just a single neuron, the output was a scalar).

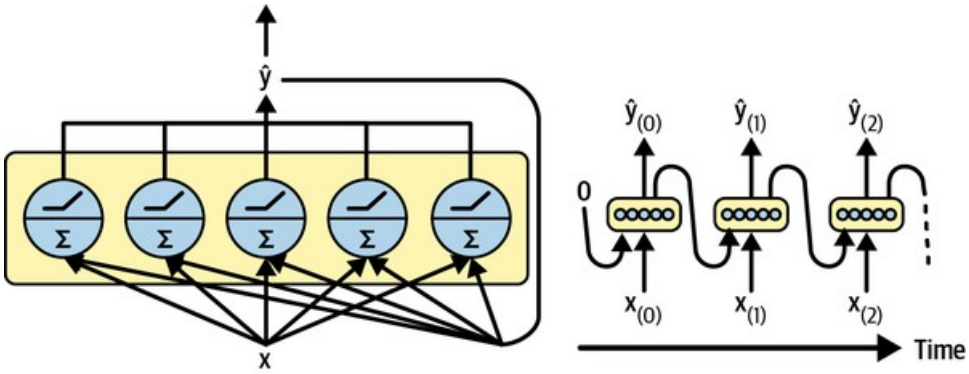


Figure 60: A layer of recurrent neurons (left) unrolled through time (right)

Each recurrent neuron has two sets of weights: one for the inputs $x_{(t)}$ and the other for the outputs of the previous time step, $\hat{y}_{(t-1)}$. Let's call these weight vectors w_x and $w_{\hat{y}}$. If we consider the whole recurrent layer instead of just one recurrent neuron, we can place all the weight vectors in two weight matrices: W_x and $W_{\hat{y}}$. The output vector of the whole recurrent layer can then be computed pretty much as we might expect, as shown in equation below, where b is the bias vector and $\phi(\cdot)$ is the activation function (e.g., ReLU).

$$\hat{\mathbf{y}}_{(t)} = \phi\left(\mathbf{W}_x{}^{\intercal}\mathbf{x}_{(t)} + \mathbf{W}_{\hat{y}}{}^{\intercal}\hat{\mathbf{y}}_{(t-1)} + \mathbf{b}\right)$$

Figure 61: Output of a recurrent layer for a single instance

Just as with feedforward neural networks, a recurrent layer's output for an entire mini-batch can also be computed in one shot by placing all the inputs at time step t into an input matrix $X_{(t)}$ (see equation below).

$$\begin{aligned}\hat{\mathbf{Y}}_{(t)} &= \phi\left(\mathbf{X}_{(t)}\mathbf{W}_x + \hat{\mathbf{Y}}_{(t-1)}\mathbf{W}_{\hat{y}} + \mathbf{b}\right)\\ &= \phi\left(\begin{bmatrix}\mathbf{X}_{(t)} & \hat{\mathbf{Y}}_{(t-1)}\end{bmatrix}\mathbf{W} + \mathbf{b}\right) \text{ with } \mathbf{W} = \begin{bmatrix}\mathbf{W}_x\\ \mathbf{W}_{\hat{y}}\end{bmatrix}\end{aligned}$$

Figure 62: Outputs of a layer of recurrent neurons for all instances in a pass [mini-batch]

In this equation:

- $\hat{Y}_{(t)}$ is an $m \times n_{\text{neurons}}$ matrix containing the layer's outputs at time step t for each instance in the mini-batch (m is the number of instances in the mini-batch and n_{neurons} is the number of neurons).
- $X_{(t)}$ is an $m \times n_{\text{inputs}}$ matrix containing the inputs for all instances (n_{inputs} is the number of input features).
- W_x is an $n_{\text{inputs}} \times n_{\text{neurons}}$ matrix containing the connection weights for the inputs of the current time step.

- $W_{\hat{y}}$ is an $n_{neurons} \times n_{neurons}$ matrix containing the connection weights for the outputs of the previous time step.
- b is a vector of size $n_{neurons}$ containing each neuron's bias term.
- The weight matrices W_x and $W_{\hat{y}}$ are often concatenated vertically into a single weight matrix W of shape $(n_{inputs} + n_{neurons}) \times n_{neurons}$ (see the second line of the equation above).
- The notation $[X_{(t)}\ \hat{Y}_{(t-1)}]$ represents the horizontal concatenation of the matrices $X_{(t)}$ and $\hat{Y}_{(t-1)}$.

Memory Cells

Since the output of a recurrent neuron at time step *t* is a function of all the inputs from previous time steps, it gets a form of memory. A part of a neural network that preserves some state across time steps is called a *memory cell* (or simply a *cell*). A single recurrent neuron, or a layer of recurrent neurons, is a very basic cell, capable of learning only short patterns (typically about 10 steps long, but this varies depending on the task).

A cell's state at time step *t*, denoted $h_{(t)}$ (the "h" stands for "hidden"), is a function of some inputs at that time step and its state at the previous time step: $h_{(t)} = f(x_{(t)}, h_{(t-1)})$. Its output at time step *t*, denoted $\hat{y}_{(t)}$, is also a function of the previous state and the current inputs. In the case of the basic cells we have discussed so far, the output is just equal to the state, but in more complex cells this is not always the case, as shown in figure below.

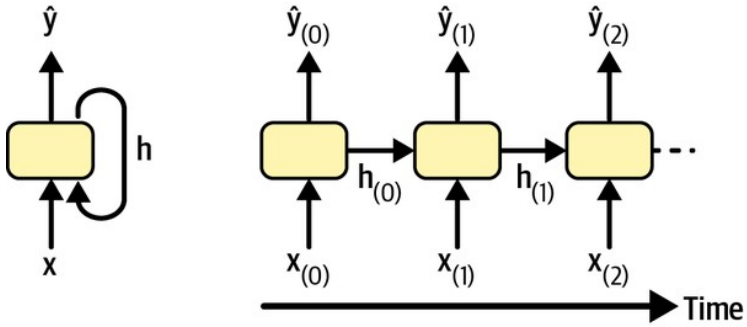


Figure 63: A cell's hidden state and its output may be different

Input and Output Sequences

Sequence-to-Sequence Network: An RNN can simultaneously take a sequence of inputs and produce a sequence of outputs (see the top-left network in figure below). This type of *sequence-to-sequence network* is useful to forecast time series, such as predicting the weather, machine translation (using an Encoder–Decoder architecture), video captioning, speech to text, music generation (or other sequence generation), identifying the chords of a song.

Sequence-to-Vector Network: Alternatively, the network is fed with a sequence of inputs and ignoring all outputs except for the last one (see the top-right network in figure below). This is a *sequence-to-vector network*. For example, the network is fed with a sequence of words corresponding to a movie review, and the network would output a sentiment score (e.g., from 0

[hate] to 1 [love]). Other examples could be classifying music samples by music genre, analyzing the sentiment of a book review, predicting what word an aphasic patient is thinking of based on readings from brain implants, predicting the probability that a user will want to watch a movie based on their watch history.

Vector-to-Sequence Network: Conversely, the network can be fed with the same input vector over and over again at each time step and letting it output a sequence (see the bottom-left network of figure below). This is a *vector-to-sequence network*. For example, the input could be an image (or the output of a CNN), and the output could be a caption for that image. Other examples are creating a music playlist based on an embedding of the current artist, generating a melody based on a set of parameters.

Encoder-Decoder Network: Lastly, we could have a sequence-to-vector network followed by a vector-to-sequence network. In this architecture, the sequence-to-vector network is called an encoder and the vector-to-sequence network is called a decoder (see the bottom-right network of figure below). For example, this could be used for translating a sentence from one language to another. The network could be fed with a sentence in one language, the encoder would convert this sentence into a single vector representation, and then the decoder would decode this vector into a sentence in another language. This two-step model, called an encoder–decoder, works much better than trying to translate on the fly with a single sequence-to-sequence RNN (like the one represented at the top left). This approach could work better because the last words of a sentence can affect the first words of the translation, so the whole sentence needs to be seen first before translating it.

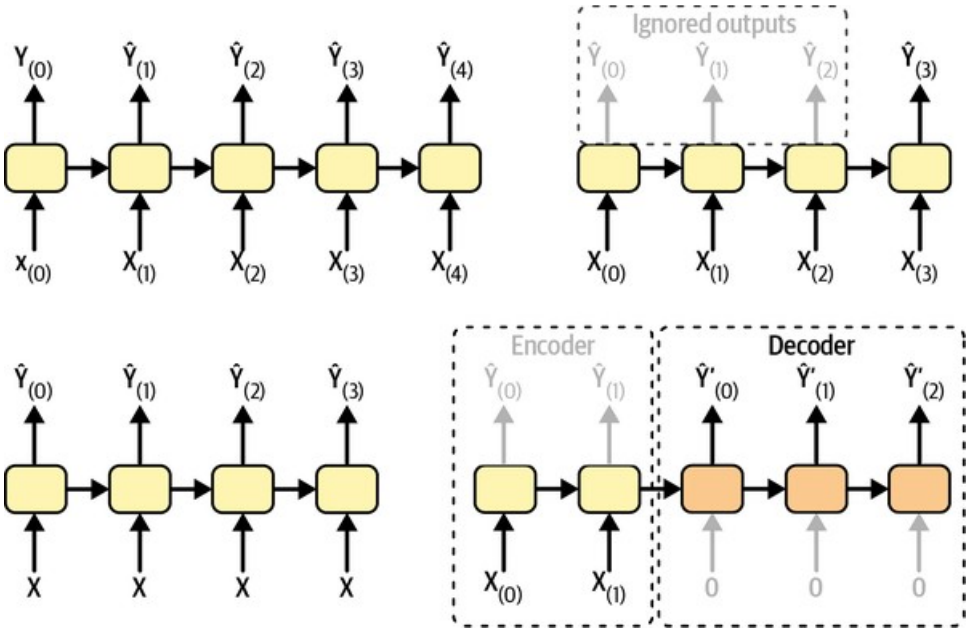


Figure 64: Sequence-to-sequence (top left), sequence-to-vector (top right), vector-to-sequence (bottom left), and encoder–decoder (bottom right) networks

Training RNNs

To train an RNN, the trick is to unroll it through time and then use regular backpropagation (see figure below). This strategy is called backpropagation through time (BPTT).

Just like in regular backpropagation, there is a first forward pass through the unrolled network (represented by the dashed arrows). Then the output sequence is evaluated using a loss function $\mathcal{L}(Y_{(0)}, Y_{(1)}, \ldots, Y_{(T)}; \hat{Y}_{(0)}, \hat{Y}_{(1)}, \ldots, \hat{Y}_{(T)})$ (where $Y_{(i)}$ is the i^{th} target, $\hat{Y}_{(i)}$ is the i^{th} prediction, and T is the max time step). It is to be noted that this loss function may ignore some outputs. For example, in a sequence-to-vector RNN, all outputs are ignored except for the very last one. In below figure, the loss function is computed based on the last three outputs only. The gradients of that loss function are then propagated backward through the unrolled network (represented by the solid arrows). In this example, since the outputs $\hat{Y}_{(0)}$ and $\hat{Y}_{(1)}$ are not used to compute the loss, the gradients do not flow backward through them; they only flow through $\hat{Y}_{(2)}$, $\hat{Y}_{(3)}$, and $\hat{Y}_{(4)}$. Moreover, since the same parameters W and b are used at each time step, their gradients will be tweaked multiple times during backprop. Once the backward phase is complete and all the gradients have been computed, BPTT can perform a gradient descent step to update the parameters (this is no different from regular backprop).

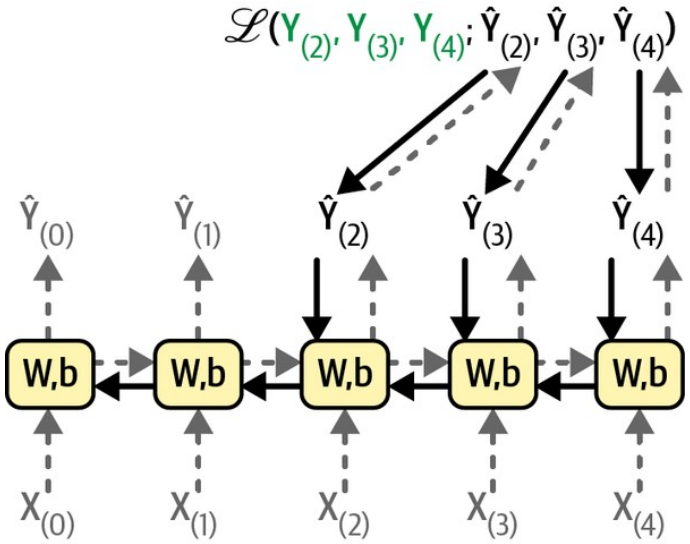


Figure 65: Backpropagation through time

Forecasting a Time Series

The following section considers a bus and rail ridership time series dataset available since 2001 and builds a model capable of forecasting the number of passengers that will ride on bus and rail the next day.

```
import pandas as pd

df = pd.read_csv(“Ridership_-_Daily_Boarding”, parse_dates=["service_date"])
df.columns = ["date", "day_type", "bus", "rail"] # shorter names
df = df.sort_values("date").set_index("date")

df.head()
```

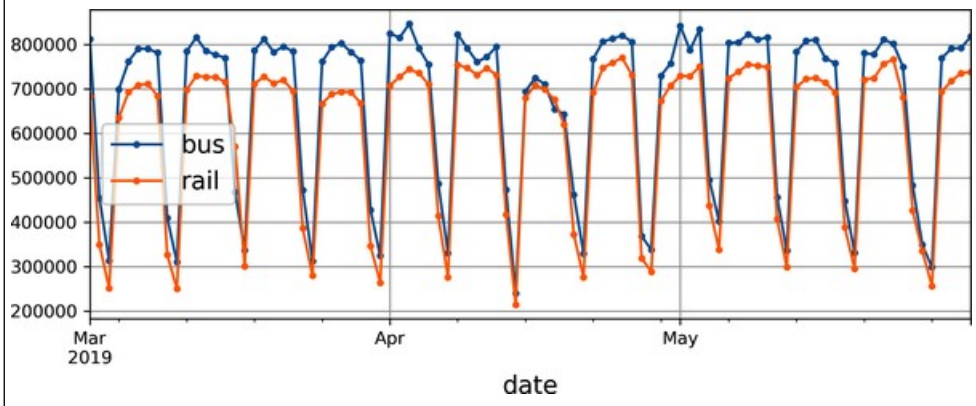
```
           day_type     bus    rail
date
2001-01-01        U  297192  126455
2001-01-02        W  780827  501952
2001-01-03        W  824923  536432
2001-01-04        W  870021  550011
2001-01-05        W  890426  557917
```

The day_type column contains W for Weekdays, A for Saturdays, and U for Sundays or holidays.

The following plots the bus and rail ridership figures over a few months in 2019, to see what it looks like (see figure below).

```
import matplotlib.pyplot as plt

df["2019-03":"2019-05"].plot(grid=True, marker=".", figsize=(8, 3.5))
plt.show()
```



This is a time series: data with values at different time steps, usually at regular intervals. More specifically, since there are multiple values per time step, this is called a *multivariate time series*. If only the bus column is looked at, it would be a *univariate time series*, with a single value per time step. Predicting future values (i.e., forecasting) is the most typical task when dealing with time series. Other tasks include imputation (filling in missing past values), classification, anomaly detection, and more.

The figure shows that a similar pattern is clearly repeated every week. This is called a weekly *seasonality*. Naive forecasting is simply copying a past value to make our forecast. Naive forecasting is often a great baseline, and it can even be tricky to beat in some cases.

To visualize these naive forecasts, the two time series are overlayed (for bus and rail) as well as the same time series lagged by one week (i.e., shifted toward the right) using dotted lines. The difference between the two (i.e., the value at time t minus the value at time $t - 7$) is also plotted; this is called differencing (see figure below).

```
diff_7 = df[["bus", "rail"]].diff(7)["2019-03":"2019-05"]
fig, axs = plt.subplots(2, 1, sharex=True, figsize=(8, 5))
df.plot(ax=axs[0], legend=False, marker=".") # original time series
df.shift(7).plot(ax=axs[0], grid=True, legend=False, linestyle=":") # lagged
```

```
diff_7.plot(ax=axs[1], grid=True, marker=".") # 7-day difference time series
plt.show()
```

The lagged time series track the actual time series very closely. When a time series is correlated with a lagged version of itself, the time series is called *autocorrelated*.

It now measures the mean absolute error over the three-month period we're arbitrarily focusing on —March, April, and May 2019—to get a rough idea.

```
diff_7.abs().mean()

bus    43915.608696
rail   42143.271739
dtype: float64
```

Naive forecasts get an MAE of about 43,916 bus riders, and about 42,143 rail riders. Mean absolute percentage error (MAPE) is computed by dividing them by the target values.

```
>>> targets = df[["bus", "rail"]]["2019-03":"2019-05"]
>>> (diff_7 / targets).abs().mean()

bus    0.082938
rail   0.089948
dtype: float64
```

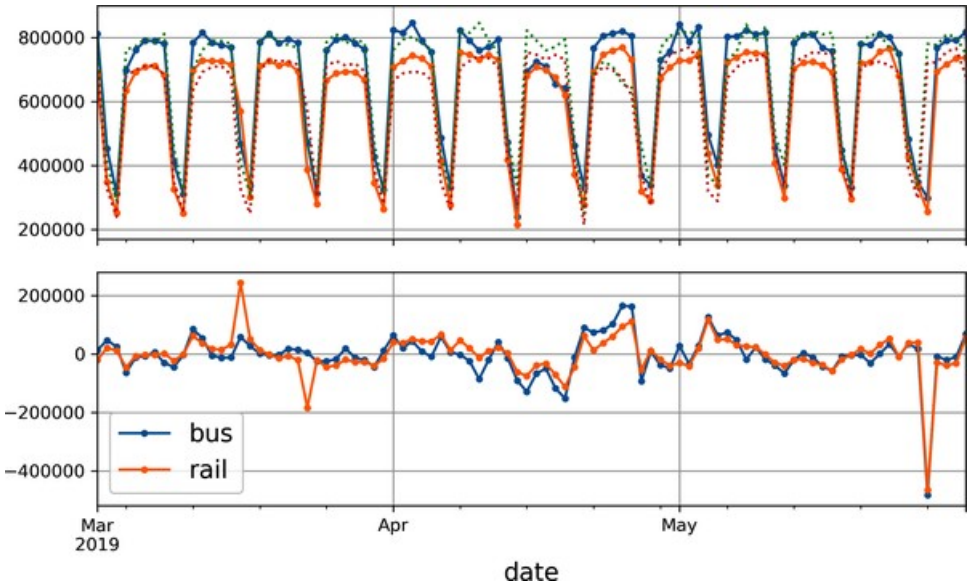


Figure 66: Time series overlaid with 7-day lagged time series (top), and difference between t

Naive forecasts give us a MAPE of roughly 8.3% for bus and 9.0% for rail.

Yearly seasonality is checked by looking at the data from 2001 to 2019 and by plotting a 12-month rolling average for each series to visualize long-term trends (see figure below):

```
period = slice("2001", "2019")
df_monthly = df.resample('M').mean()   # compute the mean for each month
rolling_average_12_months = df_monthly[period].rolling(window=12).mean()

fig, ax = plt.subplots(figsize=(8, 4))
df_monthly[period].plot(ax=ax, marker=".")
rolling_average_12_months.plot(ax=ax, grid=True, legend=False)
plt.show()
```

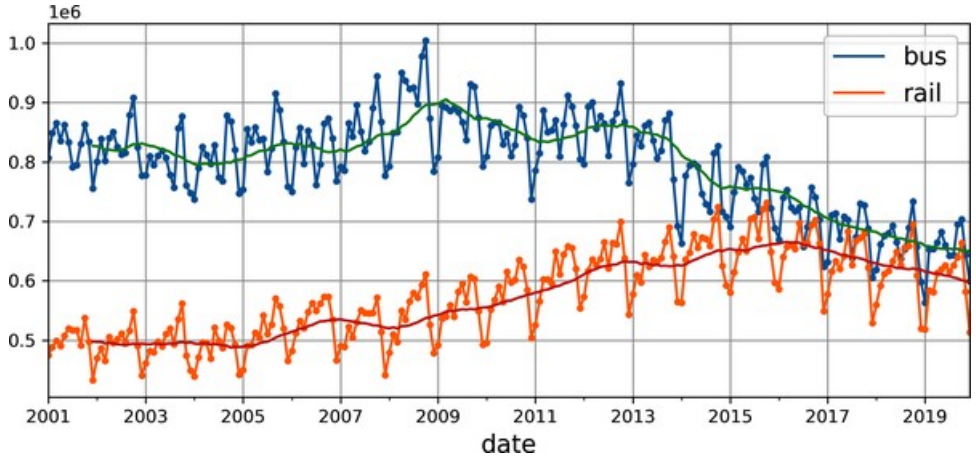


Figure 67: Yearly seasonality and long-term trends

Yearly seasonality is more visible for the rail series than the bus series. Let's check what we get if we plot the 12-month difference (see figure below)

```
df_monthly.diff(12)[period].plot(grid=True, marker=".", figsize=(8, 3))
plt.show()
```

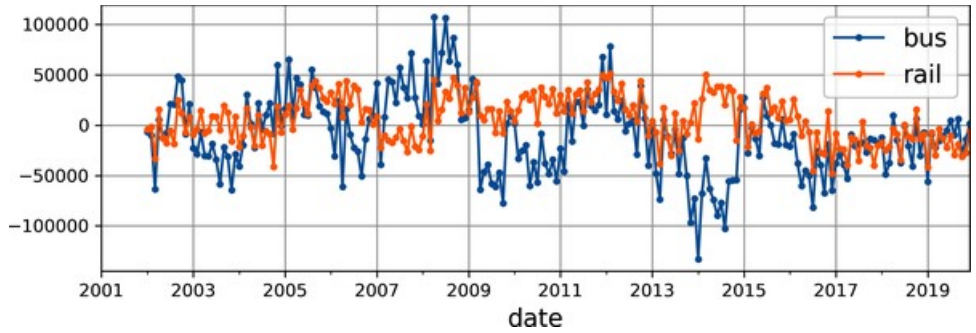


Figure 68: The 12-month difference

Differencing not only removed the yearly seasonality, but it also removed the long-term trends. For example, the linear downward trend present in the time series from 2016 to 2019 became a roughly constant negative value in the differenced time series. In fact, differencing is a common technique used to remove trend and seasonality from a time series: it's easier to study a *stationary* time series, meaning one whose statistical properties remain constant over time, without any seasonality or trends. Once accurate forecasts can be made on the differenced time series, it's easy to turn them

into forecasts for the actual time series by just adding back the past values that were previously subtracted.

Implementing a Simple RNN

The followings implements the most basic RNN, containing a single recurrent layer with just one recurrent neuron.

```
model = tf.keras.Sequential([
    tf.keras.layers.SimpleRNN(1, input_shape=[None, 1])
])
```

All recurrent layers in Keras expect 3 dimension shape inputs of [*batch size, time steps, dimensionality*], where dimensionality is 1 for univariate time series and more for multivariate time series. The `input_shape` argument ignores the first dimension (i.e., the batch size) and since recurrent layers can accept input sequences of any length, the second dimension is set to None, which means "any size". The last dimension's size is set to 1 as this being a univariate time series. This is why we specified the input shape [None, 1]: it means "univariate sequences of any length".

This model works exactly as seen earlier: the initial state $h_{(init)}$ is set to 0, and it is passed to a single recurrent neuron, along with the value of the first time step, $x_{(0)}$. The neuron computes a weighted sum of these values plus the bias term, and it applies the activation function to the result, using the hyperbolic tangent function by default. The result is the first output, y_0. In a simple RNN, this output is also the new state h_0. This new state is passed to the same recurrent neuron along with the next input value, $x_{(1)}$, and the process is repeated until the last time step. At the end, the layer just outputs the last value: in this case the sequences are 56 steps long, so the last value is y_{55}. All of this is performed simultaneously for every sequence in the batch, of which there are 32 in this case. By default, recurrent layers in Keras only return the final output. To make them return one output per time step, `return_sequences` must be set to True. It's a sequence-to-vector model. Since there's a single output neuron, the output vector has a size of 1. To build a deep sequence-to-sequence RNN using Keras, `return_sequences=True` is set for all RNN layers.

There could be the following reasons if its prediction performance is not good.

1. The RNN's memory is extremely limited: The model only has a single recurrent neuron, so the only data it can use to make a prediction at each time step is the input value at the current time step and the output value from the previous time step. Also whole model only has three parameters (two weights plus a bias term).

2. The default activation function is tanh, the recurrent layer can only output values between –1 and +1. There's no way it can predict values beyond this range that would required by some task.

Deep RNNs

It is quite common to stack multiple layers of cells, as shown in figure below. This gives you a deep RNN.

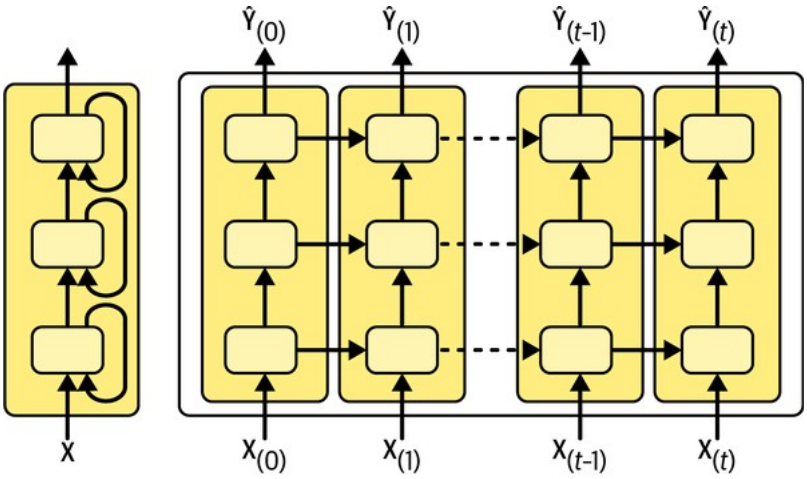


Figure 69: A deep RNN (left) unrolled through time (right)

Implementing a deep RNN with Keras is straightforward: recurrent layers are just stacked. In the following example, three `SimpleRNN` layers are used. The first two are sequence-to-sequence layers, and the last one is a sequence-to-vector layer. Finally, the Dense layer produces the model's forecast. This can be considered as a vector-to-vector layer. Finally, the Dense layer produces the model's forecast. So this model is just like the model represented in the above figure, except the outputs $\hat{Y}_{(0)}$ to $\hat{Y}_{(t-1)}$ are ignored, and there's a dense layer on top of $\hat{Y}_{(t)}$, which outputs the actual forecast.

```
deep_model = tf.keras.Sequential([
  tf.keras.layers.SimpleRNN(32, return_sequences=True, input_shape=[None, 1]),
  tf.keras.layers.SimpleRNN(32, return_sequences=True),
  tf.keras.layers.SimpleRNN(32),
  tf.keras.layers.Dense(1)
])
```

The parameter `return_sequences` should be set to `True` for all recurrent layers (except the last one, if only the last output is required).

Forecasting Several Time Steps Ahead

Prediction for several steps ahead can also be done by changing the targets appropriately.

First Option: An RNN can be trained to output one target for the time series and adding that value to the inputs, acting as if the predicted value had actually occurred; then the model would be used again to predict the following value, and so on. Downside of this approach is that if the model

makes an error at one time step, then the forecasts for the following time steps are impacted as well: the errors tend to accumulate. So, it's preferable to use this technique only for a small number of steps.

```
series = generate_time_series(1, n_steps + 10)
X_new, Y_new = series[:, :n_steps], series[:, n_steps:]
X = X_new

for step_ahead in range(10):
    y_pred_one = model.predict(X[:, step_ahead:])[:, np.newaxis, :]
    X = np.concatenate([X, y_pred_one], axis=1)

Y_pred = X[:, n_steps:]
```

Second Option: Alternatively, prediction can be done several time steps ahead in one shot. Sequence-to-vector model can be used to output *n* values instead of 1. However, the targets needs to be changed to be vectors containing the next *n* values. To do this, `timeseries_dataset_from_array()` can be used asking it to create datasets without targets (targets=None) and with longer sequences, of length seq_length + *n*. Then the datasets' map() method can be used to apply a custom function to each batch of sequences, splitting them into inputs and targets. Multivariate time series can be used as input, and prediction for the next *n* time steps would be the forecast.

```
# Helper functions

n = 14    # Number of time steps ahead to forecast

def split_inputs_and_targets(mulvar_series, ahead=n, target_col=1):
    return mulvar_series[:, :-ahead], mulvar_series[:, -ahead:, target_col]


ahead_train_ds = tf.keras.utils.timeseries_dataset_from_array(
    mulvar_train.to_numpy(),
    targets=None,
    sequence_length=seq_length + n,
    [...]                                  # the other 3 arguments are the same as earlier
).map(split_inputs_and_targets)

ahead_valid_ds = tf.keras.utils.timeseries_dataset_from_array(
    mulvar_valid.to_numpy(),
    targets=None,
    sequence_length=seq_length + n,
    batch_size=32
).map(split_inputs_and_targets)


# There should be n units instead of 1 in the output layer

ahead_model = tf.keras.Sequential([
    tf.keras.layers.SimpleRNN(32, input_shape=[None, 5]),
    tf.keras.layers.Dense(n)
])

# After training this model, the next n values are predicted at once:

X = mulvar_valid.to_numpy()[np.newaxis, :seq_length] # shape [1, 56, 5]
Y_pred = ahead_model.predict(X) # shape [1, n]
```

Handling Long Sequences

To train an RNN on long sequences, it is run over many time steps, making the unrolled RNN a very deep network. Just like any deep neural network it may suffer from the unstable gradients problem: it may take forever to train, or training may be unstable. Moreover, when an RNN processes a long sequence, it will gradually forget the first inputs in the sequence. These two problems are discussed below.

Fighting the Unstable Gradients Problem

Many of the tricks that are used in deep nets to alleviate the unstable gradients problem can also be used for RNNs: good parameter initialization, faster optimizers, dropout, and so on. However, non-saturating activation functions (e.g., ReLU) may not help as much here. In fact, they may actually lead the RNN to be even more unstable during training. Supposing gradient descent updates the weights in a way that increases the outputs slightly at the first time step, the outputs at the second time step may also be slightly increased considering the same weights are used at every time step, and those at the third, and so on until the outputs explode—and a non-saturating activation function does not prevent that. This risk could be reduced by using a smaller learning rate, or a saturating activation function like the hyperbolic tangent (that's why it's the default activation function in RNN). In much the same way, the gradients themselves can explode. The size of the gradients can be monitored using TensorBoard) and perhaps gradient clipping can be used.

It is technically possible to add a Batch Normalization (BN) layer to a memory cell so that it will be applied at each time step (both on the inputs for that time step and on the hidden state from the previous step). However, the same BN layer will be used at each time step, with the same parameters, regardless of the actual scale and offset of the inputs and hidden state. In practice, this does not yield good results as it works slightly better than nothing when applied between recurrent layers, but not within recurrent layers (i.e., horizontally). In Keras, BN between layers can be applied simply by adding a `BatchNormalization` layer before each recurrent layer, but it will slow down training, and it may not help much.

Another form of normalization often works better with RNNs: *layer normalization*. It is very similar to batch normalization, but instead of normalizing across the batch dimension, layer normalization normalizes across the features dimension. One advantage is that it can compute the required statistics on the fly, at each time step, independently for each instance. This also means that it behaves the same way during training and testing (as opposed to BN), and it does not need to use exponential moving averages to estimate the feature statistics across all instances in the training set, like BN does. Like BN, layer normalization learns a scale and an offset parameter for each input. In an RNN, it is typically used right after the linear combination of the inputs and the hidden states.

Tackling the Short-Term Memory Problem

Due to the transformations that the data goes through when traversing an RNN, some information is lost at each time step. After a while, the RNN's state contains virtually no trace of the first inputs. This can be a showstopper. To tackle this problem, various types of cells with long-term memory

have been introduced. They have proven so successful that the basic cells are not used much anymore. The most popular of these long-term memory cells – the LSTM cell is discussed below.

LSTM Cells:

The long short-term memory (LSTM) cell can be used very much like a basic cell, except it will perform much better; training will converge faster, and it will detect longer-term patterns in the data. LSTM layer uses an optimized implementation when running on a GPU, so in general it is preferable to use it.

In Keras, simply the LSTM layer can be used instead of the SimpleRNN layer as shown below.

```
model = tf.keras.Sequential([
    tf.keras.layers.LSTM(32, return_sequences=True, input_shape=[None, 5]),
    tf.keras.layers.Dense(14)
])
```

Its architecture is shown in figure below. The LSTM cell looks exactly like a regular cell, except that its state is split into two vectors: $h_{(t)}$ and $c_{(t)}$ ("c" stands for "cell"). $h_{(t)}$ can be considered as the short-term state and $c_{(t)}$ as the long-term state. The key idea is that the network can learn what to store in the long-term state, what to throw away, and what to read from it. As the long-term state $c_{(t-1)}$ traverses the network from left to right, it first goes through a forget gate, dropping some memories, and then it adds some new memories via the addition operation (which adds the memories that were selected by an *input gate*). The result $c_{(t)}$ is sent straight out, without any further transformation. So, at each time step, some memories are dropped and some memories are added. Moreover, after the addition operation, the long-term state is copied and passed through the tanh function, and then the result is filtered by the *output gate*. This produces the short-term state $h_{(t)}$ (which is equal to the cell's output for this time step, $y_{(t)}$). Now the followings discusses where new memories come from and how the gates work.

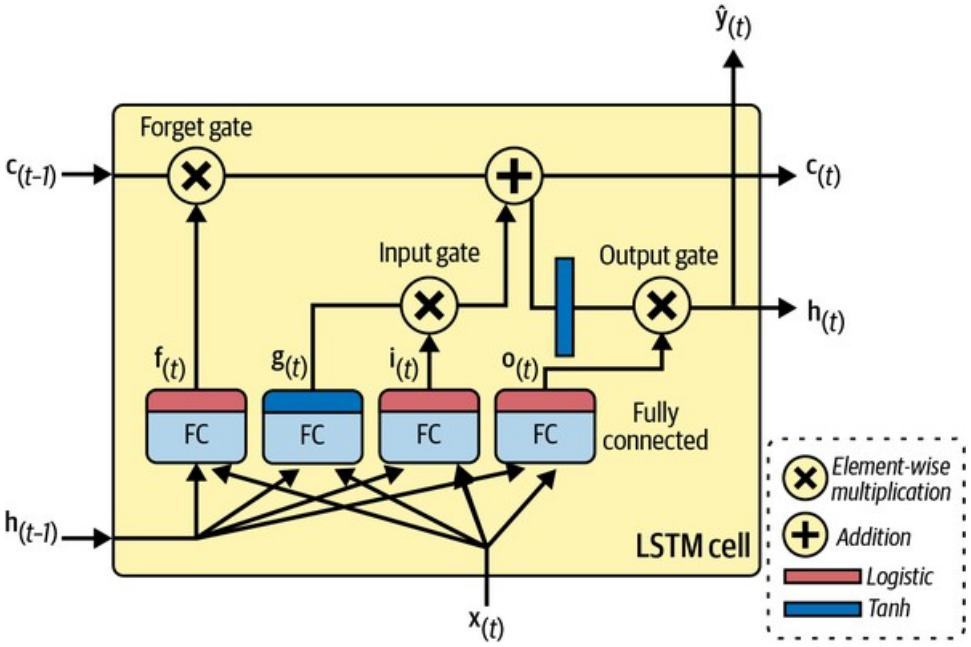


Figure 70: An LSTM cell

First, the current input vector $x_{(t)}$ and the previous short-term state $h_{(t-1)}$ are fed to four different fully connected layers. They all serve a different purpose:

- The main layer is the one that outputs $g_{(t)}$. It has the usual role of analyzing the current inputs $x_{(t)}$ and the previous (short-term) state $h_{(t-1)}$. In a basic cell, there is nothing other than this layer, and its output goes straight out to $y_{(t)}$ and $h_{(t)}$. But in an LSTM cell, this layer's output does not go straight out; instead its most important parts are stored in the long-term state (and the rest is dropped).
- The three other layers are *gate controllers*. Since they use the logistic activation function, the outputs range from 0 to 1. The gate controllers' outputs are fed to element-wise multiplication operations: if they output 0s they close the gate, and if they output 1s they open it. Specifically:
  - The *forget gate* (controlled by $f_{(t)}$) controls which parts of the long-term state should be erased.
  - The input gate (controlled by $i_{(t)}$) controls which parts of $g_{(t)}$ should be added to the long-term state.
  - Finally, the output gate (controlled by $o_{(t)}$) controls which parts of the long-term state should be read and output at this time step, both to $h_{(t)}$ and to $y_{(t)}$.

In short, an LSTM cell can learn to recognize an important input (that's the role of the input gate), store it in the long-term state, preserve it for as long as it is needed (that's the role of the forget gate), and extract it whenever it is needed. This explains why these cells have been amazingly successful at capturing long-term patterns in time series, long texts, audio recordings, and more.

The below equation summarizes how to compute the cell's long-term state, its short-term state, and its output at each time step for a single instance.

$$
\begin{aligned}
\mathbf{i}_{(t)} &= \sigma\left(\mathbf{W}_{xi}{}^{\intercal}\mathbf{x}_{(t)} + \mathbf{W}_{hi}{}^{\intercal}\mathbf{h}_{(t-1)} + \mathbf{b}_i\right) \\
\mathbf{f}_{(t)} &= \sigma\left(\mathbf{W}_{xf}{}^{\intercal}\mathbf{x}_{(t)} + \mathbf{W}_{hf}{}^{\intercal}\mathbf{h}_{(t-1)} + \mathbf{b}_f\right) \\
\mathbf{o}_{(t)} &= \sigma\left(\mathbf{W}_{xo}{}^{\intercal}\mathbf{x}_{(t)} + \mathbf{W}_{ho}{}^{\intercal}\mathbf{h}_{(t-1)} + \mathbf{b}_o\right) \\
\mathbf{g}_{(t)} &= \tanh\left(\mathbf{W}_{xg}{}^{\intercal}\mathbf{x}_{(t)} + \mathbf{W}_{hg}{}^{\intercal}\mathbf{h}_{(t-1)} + \mathbf{b}_g\right) \\
\mathbf{c}_{(t)} &= \mathbf{f}_{(t)} \otimes \mathbf{c}_{(t-1)} + \mathbf{i}_{(t)} \otimes \mathbf{g}_{(t)} \\
\mathbf{y}_{(t)} &= \mathbf{h}_{(t)} = \mathbf{o}_{(t)} \otimes \tanh\left(\mathbf{c}_{(t)}\right)
\end{aligned}
$$

Equation 13: LSTM computations

In this equation:

- W_{xi}, W_{xf} , W_{xo}, and W_{xg} are the weight matrices of each of the four layers for their connection to the input vector $x_{(t)}$.
- W_{hi}, W_{hf} , W_{ho}, and W_{hg} are the weight matrices of each of the four layers for their connection to the previous short-term state $h_{(t-1)}$.
- b_i, b_f , b_o, and b_g are the bias terms for each of the four layers. Note that TensorFlow initializes b_f to a vector full of 1s instead of 0s. This prevents forgetting everything at the beginning of training.

GRU Cells:

One particularly popular variant of the LSTM cell is *gated recurrent unit* (GRU) cell.

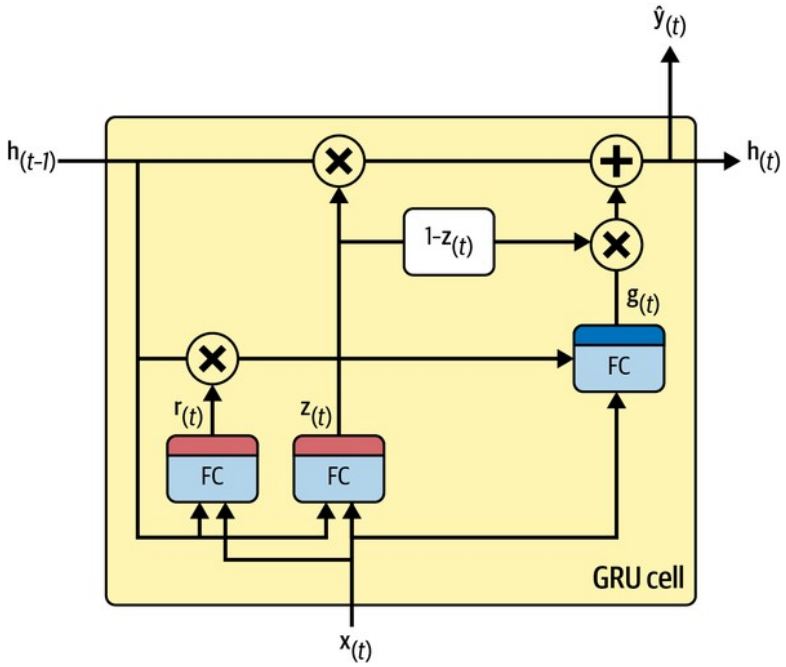


Figure 71: GRU cell

The GRU cell is a simplified version of the LSTM cell, and it seems to perform just as well. These are the main simplifications:

- Both state vectors are merged into a single vector $h_{(t)}$.
- A single gate controller $z_{(t)}$ controls both the forget gate and the input gate. If the gate controller outputs a 1, the forget gate is open (= 1) and the input gate is closed (1 – 1 = 0). If it outputs a 0, the opposite happens. In other words, whenever a memory must be stored, the location where it will be stored is erased first. This is actually a frequent variant to the LSTM cell in and of itself.
- There is no output gate; the full state vector is output at every time step. However, there is a new gate controller $r_{(t)}$ that controls which part of the previous state will be shown to the main layer ($g_{(t)}$).

Below equation summarizes how to compute the cell's state at each time step for a single instance.

$$
\begin{aligned}
\mathbf{z}_{(t)} &= \sigma\left(\mathbf{W}_{xz}{}^{\intercal}\mathbf{x}_{(t)} + \mathbf{W}_{hz}{}^{\intercal}\mathbf{h}_{(t-1)} + \mathbf{b}_z\right) \\
\mathbf{r}_{(t)} &= \sigma\left(\mathbf{W}_{xr}{}^{\intercal}\mathbf{x}_{(t)} + \mathbf{W}_{hr}{}^{\intercal}\mathbf{h}_{(t-1)} + \mathbf{b}_r\right) \\
\mathbf{g}_{(t)} &= \tanh\left(\mathbf{W}_{xg}{}^{\intercal}\mathbf{x}_{(t)} + \mathbf{W}_{hg}{}^{\intercal}\left(\mathbf{r}_{(t)} \otimes \mathbf{h}_{(t-1)}\right) + \mathbf{b}_g\right) \\
\mathbf{h}_{(t)} &= \mathbf{z}_{(t)} \otimes \mathbf{h}_{(t-1)} + \left(1 - \mathbf{z}_{(t)}\right) \otimes \mathbf{g}_{(t)}
\end{aligned}
$$

Equation 14: GRU computations

Keras provides a `tf.keras.layers.GRU` layer: using it is just a matter of replacing SimpleRNN or LSTM with GRU.

Exercises

1. Explain recurrent neuron and recurrent layer and discuss how they predicts on against a sequence.
2. Can you think of a few applications for a sequence-to-sequence RNN? What about a sequence-to-vector RNN, and a vector-to-sequence RNN? Can you explain encoder-decoder network and discuss how it could be useful in natural language translation? Draw diagram for each mentioned network.
3. How does training takes place in RNN?
4. How many dimensions must the inputs of an RNN layer have? What does each dimension represent? What about its outputs? How does is simple RNN implemented?
5. If you want to build a deep sequence-to-sequence RNN, which RNN layers should have return_sequences=True? What about a sequence-to-vector RNN?
6. Suppose you have a daily univariate time series, and you want to forecast the next seven days. Which RNN architecture should you use?

7. What are the main difficulties when training RNNs? How can you handle them?
8. Explain Deep RNN and how it can be implemented using Simple RNN.
9. Discuss two options to implement forecasting several time steps ahead using RNN.
10. What are the two major problems that RNN faces to handle long sequences? How to fight unstable gradient problem and short-term memory problem related to RNN.
11. Can you explain the LSTM and draw the cell's architecture?
12. Can you explain the GRU and draw the cell's architecture?

Natural Language Processing (NLP)

A common approach for natural language tasks is to use recurrent neural networks (RNNs). For example, to generate some original text, a *character* RNN can be trained to predict the next character in a sentence. A *stateless* RNN learns on random portions of text at each iteration, without any information on the rest of the text whereas a *stateful* RNN preserves the hidden state between training iterations and continues reading where it left off, allowing it to learn longer patterns. RNN can also be used perform sentiment analysis (e.g., reading movie reviews and extracting the rater's feeling about the movie) by treating sentences as sequences of words, rather than characters. RNNs can also be used to build an Encoder–Decoder architecture capable of performing neural machine translation (NMT).

The performance of an RNN-based Encoder–Decoder architecture can be boosted using *attention mechanism* – a neural network components that learns to select the part of the inputs that the rest of the model should focus on at each time step. Attention-only architectures called the *Transformers*. Few of the incredibly powerful language models that are based on Transformers are GPT-2 and BERT.

Generating Text Using a Character RNN

An RNN can be trained to predict the next character in a sentence. This *Char-RNN* can then be used to generate novel text, one character at a time, and the model can learn words, grammar, proper punctuation, and more, just by learning to predict the next character in a sentence. Major steps are described below.

Creating the Training Dataset

It first downloads the data – Shakespear's work for example.

```
filepath = keras.utils.get_file("shakespeare.txt", <dataset_url>)

with open(filepath) as f:
    shakespeare_text = f.read()
```

Next, every character is encoded as an integer. Keras's `Tokenizer` class makes this task simple. First the tokenizer is fit to the text and it finds all the characters used in the text and maps each of them to a different character ID, from 1 to the number of distinct characters (it does not start at 0).

```
tokenizer = keras.preprocessing.text.Tokenizer(char_level=True)
tokenizer.fit_on_texts([shakespeare_text])
```

The tokenizer converts the text to lowercase by default. Now the tokenizer can encode a sentence (or a list of sentences) to a list of character IDs and back. The following encodes the full text so each character is represented by its ID (1 is subtracted to get IDs from 0 rather than from 1).

```
[encoded] = np.array(tokenizer.texts_to_sequences([shakespeare_text])) - 1
```

Splitting a Sequential Dataset

The dataset needs to be split into a training set, a validation set, and a test set. All the characters in the text can't just be shuffled because it's a sequential dataset. For time series data, it is very important to avoid any overlap between the training set, the validation set, and the test set. It is fine if the first 90% of the text is taken for the training set, then the next 5% is taken for the validation set, and the final 5% is taken for the test set. It would also be a good idea to leave a gap between these sets to avoid the risk of a paragraph overlapping over two sets.

For this example, first 90% of the text is taken for the training set (keeping the rest for the validation set and the test set), and create a `tf.data.Dataset` that will return each character one by one from this set.

```
train_size = dataset_size * 90 // 100
dataset = tf.data.Dataset.from_tensor_slices(encoded[:train_size])
```

Chopping the Sequential Dataset into Multiple Windows

The training set now consists of a single sequence of over a million characters, so instead of training the neural network directly on it, the dataset's `window()` method is used to convert this long sequence of characters into many smaller windows of text. Every instance in the dataset will be a fairly short substring of the whole text, and the RNN will be unrolled only over the length of these substrings. This is called *truncated backpropagation through time*.

```
n_steps = 100    # it should be larger for RNN to learn pattern

window_length = n_steps + 1     # target = input shifted 1 character ahead
dataset = dataset.window(window_length, shift=1, drop_remainder=True)
```

To get the largest possible training set `shift=1` is set so that the first window contains characters 0 to 100, the second contains characters 1 to 101, and so on. To ensure that all windows are exactly 101 characters long, `drop_remainder` is set to `True`.

The `window()` method creates a *nested dataset* that contains windows, each of which is also represented as a dataset. As the model in this example will expect tensors as input, not datasets, `flat_map()` method is called to converts a nested dataset into a flat dataset.

```
dataset = dataset.flat_map(lambda window: window.batch(window_length))
```

Now the dataset contains consecutive windows of 101 characters each. Since Gradient Descent works best when the instances in the training set are independent and identically distributed, these windows get shuffled. Then the windows are batched and the inputs are separated (the first 100 characters) from the target (the last character).

```
batch_size = 32
dataset = dataset.shuffle(10000).batch(batch_size)
dataset = dataset.map(lambda windows: (windows[:, :-1], windows[:, 1:]))
```

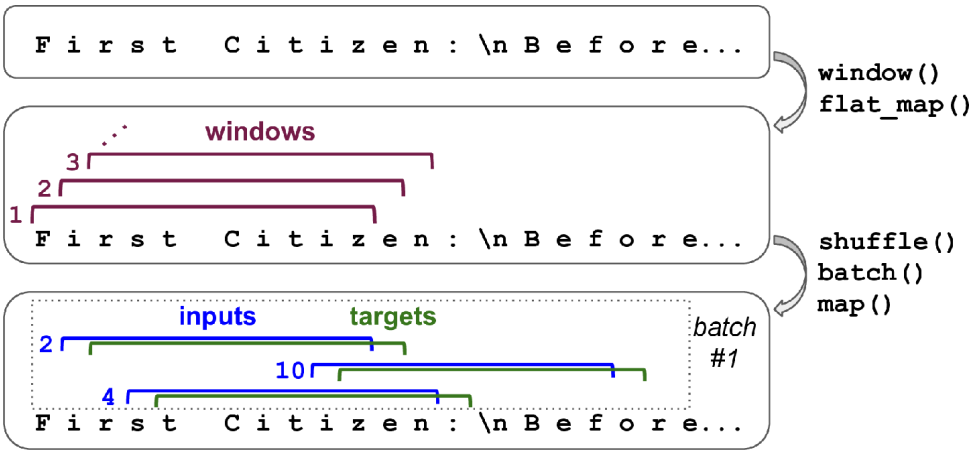


Figure 72: Preparing a dataset of shuffled windows

Categorical input features should generally be encoded, usually as one-hot vectors or as embeddings. Here, each character is encoded using a one-hot vector because there are fairly few distinct characters (only 39). Then prefetching is added.

```
dataset = dataset.map(
    lambda X_batch, Y_batch: (tf.one_hot(X_batch, depth=max_id), Y_batch))

dataset = dataset.prefetch(1)
```

Building and Training the Char-RNN Model

To predict the next character based on the previous 100 characters, an RNN with 2 GRU layers of 128 units each and 20% dropout on both the inputs (drop out) and the hidden states (recurrent_dropout) are used. These hyperparameters can be tweaked later, if needed. The output layer is a time-distributed Dense layer. This time this layer must have 39 units (max_id) because there are 39 distinct characters in the text, and we want to output a probability for each possible character (at each time step). The output probabilities should sum up to 1 at each time step, so the

softmax activation function can be applied to the outputs of the Dense layer. Then the model is compiled using the "`sparse_categorical_crossentropy`" loss and an Adam optimizer. Finally, the model is trained for several epochs.

```
model = keras.models.Sequential([
    keras.layers.GRU(128, return_sequences=True, input_shape=[None, max_id],
        dropout=0.2, recurrent_dropout=0.2),
    keras.layers.GRU(128, return_sequences=True,
        dropout=0.2, recurrent_dropout=0.2),
    keras.layers.TimeDistributed(keras.layers.Dense(max_id,
        activation="softmax"))
])

model.compile(loss="sparse_categorical_crossentropy", optimizer="adam")
history = model.fit(dataset, epochs=20)
```

Using the Char-RNN Model

To feed it some text to the model to predict the next character, the text first need to be preprocessed and and it is done over a helper function.

```
def preprocess(texts):
    X = np.array(tokenizer.texts_to_sequences(texts)) - 1
    return tf.one_hot(X, max_id)
```

The mode now predicts the next letter in some text.

```
X_new = preprocess(["How are yo"])
Y_pred = model.predict_classes(X_new)
tokenizer.sequences_to_texts(Y_pred + 1)[0][-1] # 1st sentence, last char 'u'
```

The model guessed right – the last character is ‘u’.

Generating Fake Text

To generate new text using the Char-RNN model, it could be fed with some text, and the model predicts the most likely next letter, adding that at the end of the text, then giving the extended text to the model to guess the next letter, and so on. But in practice this often leads to the same words being repeated over and over again. Instead, the next character can be picked randomly, with a probability equal to the estimated probability, using TensorFlow’s `tf.random.categorical()` function to generate more diverse and interesting text. The `categorical()` function samples random class indices, given the class log probabilities (logits). To have more control over the diversity of the generated text, the logits can be divided by a number called the *temperature*, which can be tweaked as required: a temperature close to 0 will favor the high probability characters, while a very high temperature will give all characters an equal probability. The following `next_char()` function uses this approach to pick the next character to add to the input text.

```
def next_char(text, temperature=1):
```

```
    X_new = preprocess([text])
    y_proba = model.predict(X_new)[0, -1:, :]
    rescaled_logits = tf.math.log(y_proba) / temperature
    char_id = tf.random.categorical(rescaled_logits, num_samples=1) + 1
    return tokenizer.sequences_to_texts(char_id.numpy())[0]
```

Next, the below function repeatedly calls `next_char()` to get the next character and append it to the given text.

```
def complete_text(text, n_chars=50, temperature=1):
    for _ in range(n_chars):
    text += next_char(text, temperature)
    return text
```

Now, the model generates some text against different temperatures.

```
print(complete_text("t", temperature=0.2))
the belly the great and who shall be the belly the

print(complete_text("w", temperature=1))
thing? or why you gremio.
who make which the first

print(complete_text("w", temperature=2))
th no cce:
yeolg-hormer firi. a play asks.
fol rusb
```

To generate more convincing text, the followings can be tried.

1. Using more GRU layers and more neurons per layer, train for longer, and add some regularization.
2. Making the model capable of learning patterns longer than n_steps (which is just 100 characters in this example) by making this window larger.
3. Trying out a stateful RNN.

Stateful RNN and Its Difference with Stateless RNN

At each training iteration a stateless RNNs model starts with a hidden state full of zeros, then it updates this state at each time step, and after the last time step, it throws it away, as it is not needed anymore.

A stateful RNN, on the other hand, preserves this final state after processing one training batch and uses it as the initial state for the next training batch. This way the model can learn long-term patterns despite only backpropagating through short sequences. A stateful RNN only makes sense if each input sequence in a batch starts exactly where the corresponding sequence in the previous batch left off. So the first thing to build a stateful RNN is to use sequential and non-overlapping input sequences (rather than the shuffled and overlapping sequences). When creating the Dataset, `shift=n_steps` (instead of `shift=1`) is set when calling the `window()` method. Moreover,

`shuffle()` method must not be called. Batching is much harder when preparing a dataset for a stateful RNN than it is for a stateless RNN. Indeed, if a call to `batch(32)` is made, then 32 consecutive windows would be put in the same batch, and the following batch would not continue each of these window where it left off. The first batch would contain windows 1 to 32 and the second batch would contain windows 33 to 64, so if considered, say, the first window of each batch (i.e., windows 1 and 33), it can be seen that they are not consecutive. The simplest solution to this problem is to just use "batches" containing a single window.

```
dataset = tf.data.Dataset.from_tensor_slices(encoded[:train_size])
dataset = dataset.window(window_length, shift=n_steps, drop_remainder=True)
dataset = dataset.flat_map(lambda window: window.batch(window_length))
dataset = dataset.batch(1)
dataset = dataset.map(lambda windows: (windows[:, :-1], windows[:, 1:]))
dataset = dataset.map(
    lambda X_batch, Y_batch: (tf.one_hot(X_batch, depth=max_id), Y_batch))
dataset = dataset.prefetch(1)
```

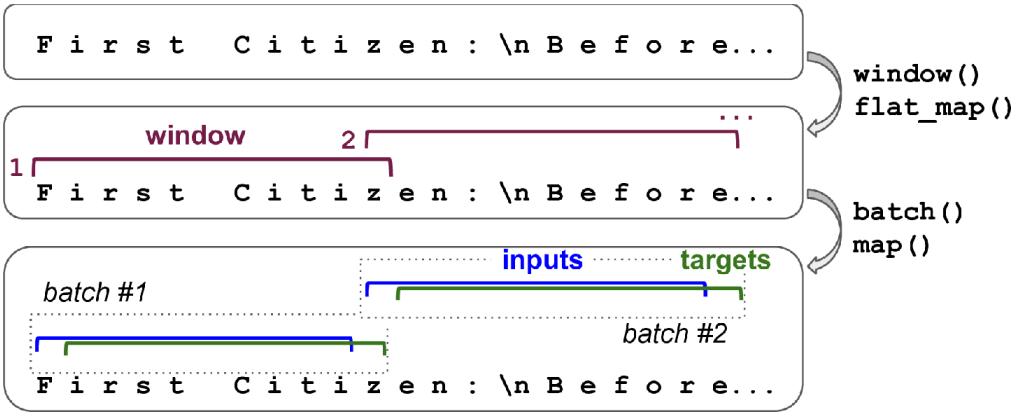


Figure 73: Preparing a dataset of consecutive sequence fragments for a stateful RNN

Now, to create the stateful RNN, first, `stateful=True` is set when creating every recurrent layer. Second, the stateful RNN needs to know the batch size (since it will preserve a state for each input sequence in the batch), so `batch_input_shape` argument must be set in the first layer. The second dimension is left unspecified, since the inputs could have any length.

```
model = keras.models.Sequential([
    keras.layers.GRU(128, return_sequences=True, stateful=True,
        dropout=0.2, recurrent_dropout=0.2,
        batch_input_shape=[batch_size, None, max_id]),
    keras.layers.GRU(128, return_sequences=True, stateful=True,
        dropout=0.2, recurrent_dropout=0.2),
    keras.layers.TimeDistributed(keras.layers.Dense(max_id,
        activation="softmax"))
])
```

At the end of each epoch, the states are reset before going back to the beginning of the text. For this, a small callback can be used.

```
class ResetStatesCallback(keras.callbacks.Callback):
    def on_epoch_begin(self, epoch, logs):
        self.model.reset_states()
```

And the model is compiled and fitted (for more epochs, because each epoch is much shorter than earlier, and there is only one instance per batch).

```
model.compile(loss="sparse_categorical_crossentropy", optimizer="adam")
model.fit(dataset, epochs=50, callbacks=[ResetStatesCallback()])
```

After this model is trained, it will only be possible to use it to make predictions for batches of the same size as were used during training. To avoid this restriction, the stateful model's weights can be copied to a new identical stateless model.

Sentiment Analysis

RNN can also be used perform sentiment analysis (e.g., reading movie reviews and extracting the rater's feeling about the movie) by treating sentences as sequences of words, rather than characters. The following discussion considers Internet Movie Database (IMDb) – a dataset that consists of 50,000 movie reviews in English (25,000 for training, 25,000 for testing), along with a simple binary target for each review indicating whether the sentiment is negative (0) or positive (1). Keras provides a simple function to load it.

```
(X_train, y_train), (X_test, y_test) = keras.datasets.imdb.load_data()

X_train[0][:10]    # shows the first few words by their indexes
[1, 14, 22, 16, 43, 530, 973, 1622, 1385, 65]
```

The dataset is already preprocessed: `X_train` consists of a list of reviews, each of which is represented as a NumPy array of integers, where each integer represents a word. All punctuation was removed, and then words were converted to lowercase, split by spaces, and finally indexed by frequency (so low integers correspond to frequent words). The integers 0, 1, and 2 are special: they represent the padding token, the start-of-sequence (SSS) token, and unknown words, respectively.

In a real project, the text needs to be preprocessed. This can be done using the same `Tokenizer` class with `char_level=False` (which is the default). When encoding words, it filters out a lot of characters, including most punctuation, line breaks, and tabs. Most importantly, it uses spaces to identify word boundaries. If the model needs to be deployed to a mobile device or a web browser, and writing a different preprocessing function every time is not expected, then the processing can be included in the model itself using TensorFlow operations.

The model might not need to know all the words in the dictionary to get good performance, so its vocabulary can be truncated, for example, by keeping only the 10,000 most common words.

Now, a preprocessing step needs to be added to replace each word with its ID and this can be done by creating a lookup table for this.

Next, the final training set gets created. The reviews are batched, then converted into short sequences of words using the preprocessing function, and then these words are encoded using lookup table, and finally the next batch is prefetched. Then the model is created and trained.

```
embed_size = 128

model = keras.models.Sequential([
    keras.layers.Embedding(vocab_size + num_oov_buckets, embed_size,
        input_shape=[None]),
    keras.layers.GRU(128, return_sequences=True),
    keras.layers.GRU(128),
    keras.layers.Dense(1, activation="sigmoid")
])

model.compile(loss="binary_crossentropy", optimizer="adam", metrics=["accuracy"])
history = model.fit(train_set, epochs=5)
```

The first layer is an `Embedding` layer, which will convert word IDs into embeddings. Whereas the inputs of the model will be 2D tensors of shape [batch size, time steps], the output of the `Embedding` layer will be a 3D tensor of shape [batch size, time steps, embedding size].

The rest of the model is composed of two `GRU` layers, with the second one returning only the output of the last time step. The output layer is just a single neuron using the sigmoid activation function to output the estimated probability that the review expresses a positive sentiment regarding the movie. Then the model is compiled and fitted on the dataset prepared earlier, for a few epochs.

Masking

As it stands, the model will need to learn that the padding tokens should be ignored to focus on the data that actually matters. This is simply done by adding `mask_zero=True` when creating the `Embedding` layer. This means that padding tokens (whose ID is 0) will be ignored by all downstream layers. All layers that receive the mask must support masking. This includes all recurrent layers, as well as the `TimeDistributed` layer and a few other layers. Any layer that supports masking must have a `supports_masking` attribute equal to `True`.

Reusing Pretrained Embeddings for Performance Improvisation

It’s often impressive that model learns useful word embeddings based on just few hundreds training dataset. The embeddings would be even better on larger dataset to train on! Perhaps word embeddings trained on some other large text corpus (e.g., Wikipedia articles) can be reused, even if it is not composed for the same task in hand. The TensorFlow Hub project makes it easy to reuse pretrained model components in the target models. These model components are called modules. Next, the dataset is just loaded without any need for preprocessingt (except for batching and prefetching) and the model is trained directly.

```
datasets, info = tfds.load("imdb_reviews", as_supervised=True, with_info=True)
train_size = info.splits["train"].num_examples
batch_size = 32
train_set = datasets["train"].batch(batch_size).prefetch(1)
history = model.fit(train_set, epochs=5)
```

An Encoder–Decoder Network for Neural Machine Translation

An encoder-decoder network can be used for natural language translation. The following discusses how a simple neural machine translation model translates English sentences to French (see figure below).

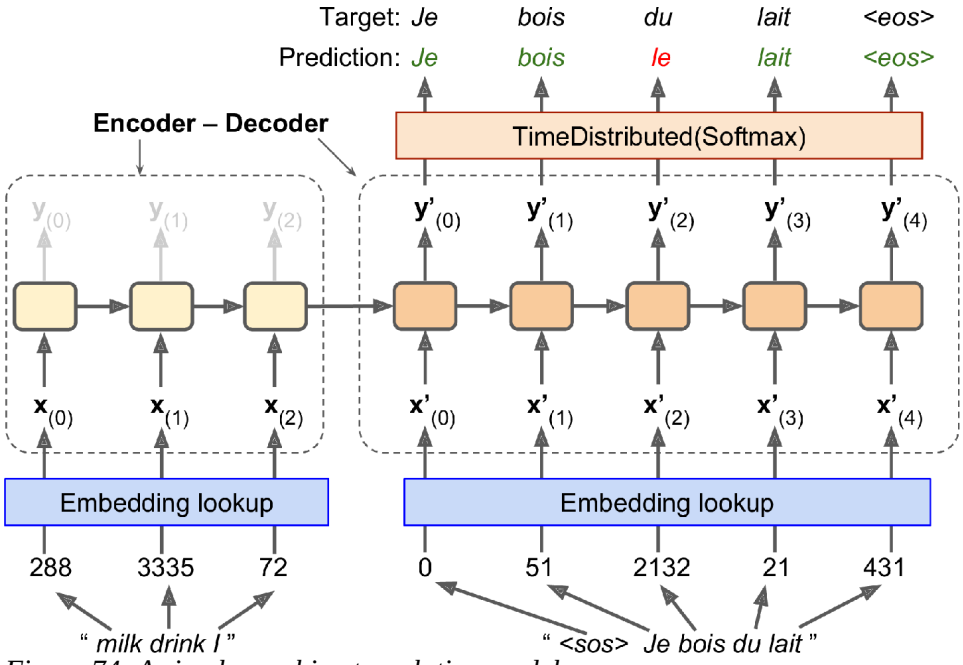


Figure 74: A simple machine translation model

In short, the English sentences are fed to the encoder, and the decoder outputs the French translations. Note that the French translations are also used as inputs to the decoder, but shifted back by one step. In other words, the decoder is given as input the word that it should have output at the previous step (regardless of what it actually output). For the very first word, it is given the start-of-sequence (SOS) token. The decoder is expected to end the sentence with an end-of-sequence (EOS) token.

The English sentences are reversed before they are fed to the encoder. For example, "I drink milk" is reversed to "milk drink I." This ensures that the beginning of the English sentence will be fed last to the encoder, which is useful because that's generally the first thing that the decoder needs to translate.

Each word is initially represented by its ID (e.g., 288 for the word "milk"). Next, an embedding layer returns the word embedding. These word embeddings are what is actually fed to the encoder and the decoder.

At each step, the decoder outputs a score for each word in the output vocabulary (i.e., French), and then the softmax layer turns these scores into probabilities. The word with the highest probability is output. This is very much like a regular classification task, so the model can be trained using the "`sparse_categorical_crossentropy`" loss.

Note that at inference time (after training), the target sentence will not be available to feed to the decoder. Instead, the decoder is simply fed with the word that it output at the previous step, as shown in figure below (this will require an embedding lookup that is not shown in the diagram).

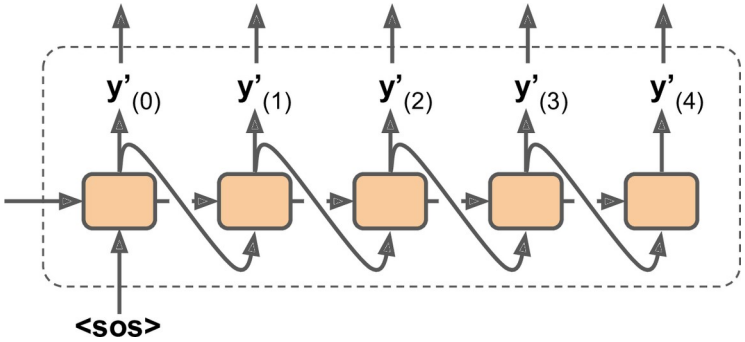


Figure 75: Feeding the previous output word as input at inference time

Few of the notable points regarding neural language translation implementations are as follows.

- **Handling Variable-length Input Sequence:** Natural language sentence lengths vary. Since regular tensors have fixed shapes, they can only contain sentences of the same length. One of ways is to use masking to handle this. But, if the sentences have very different lengths, they can't just be cropped because expectation will be for the full translation, not for the cropped translation. So one of the solutions could be to group sentences into buckets of similar lengths (e.g., a bucket for the 1- to 6-word sentences, another for the 7- to 12-word sentences, and so on) and use padding for the shorter sequences to ensure all sentences in a bucket have the same length. For example, shorter sentence "I drink milk" becomes "<pad> <pad> <pad> milk drink I."
- **Handling Variable-length Output Sequence by Ignoring Outputs Past the EOS Token:** Any output past the EOS token should be ignored, so these tokens should not contribute to the loss (they must be masked out). For example, if the model outputs "Je bois du lait <eos> oui," the loss for the last word should be ignored. The code that will use the model should ignore tokens beyond the end of the sequence. But generally the length of the output sequence is not known ahead of time, so the solution is to train the model so that it outputs an end-of-sequence token at the end of each sequence.
- **Applying Sampled Softmax Technique to Handle Output Word Probability for Large Vocabulary:** When the output vocabulary is large (say, 50,000 words), outputting a probability for each and every possible word would be terribly slow. In this case the decoder would then output 50,000-dimensional vectors, and then computing the softmax function over such a large vector would be very computationally intensive. To avoid this, one solution is to look only at the logits output by the model for the correct word and for a random sample of incorrect words, then compute an approximation of the loss based only on these logits. This is called *sampled softmax* technique and can be used during training and the normal softmax function can be used at inference time (sampled softmax cannot be used at inference time because it requires knowing the target).

Bidirectional RNNs

A each time step, a regular recurrent layer only looks at past and present inputs before generating its output. In other words, it is "causal," meaning it cannot look into the future. This type of RNN makes sense when forecasting time series, but for many NLP tasks, such as Neural Machine Translation, it is often preferable to look ahead at the next words before encoding a given word. For example, consider the phrases "the Queen of the United Kingdom," "the queen of hearts," and "the queen bee": to properly encode the word "queen," it is needed to look ahead. To implement this, run two recurrent layers on the same inputs, one reading the words from left to right and the other reading them from right to left. Then simply combine their outputs at each time step, typically by concatenating them. This is called a bidirectional recurrent layer (see figure below).

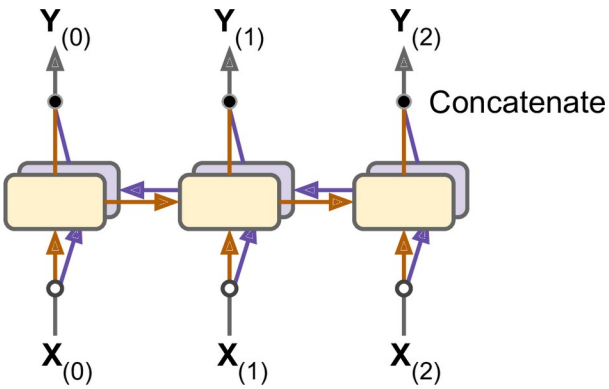


Figure 76: A bidirectional recurrent layer

To implement a bidirectional recurrent layer in Keras, a recurrent layer is wrapped in a `keras.layers.Bidirectional` layer. For example, the following code creates a bidirectional `GRU` layer. The `Bidirectional` layer will create a clone of the `GRU` layer (but in the reverse direction), and it will run both and concatenate their outputs. So although the `GRU` layer has 10 units, the `Bidirectional` layer will output 20 values per time step.

```
keras.layers.Bidirectional(keras.layers.GRU(10, return_sequences=True))
```

Beam Search and Boosting Encoder-Decoder Model Performance

Encoder-Decoder model often makes mistakes in translating language even though it is not absurd for a machine. For example, it is expected that translating French sentence "Comment vas-tu?" to English would produce "How are you?", but it may produce "How will you?" if are many training sentences such as "Comment vas-tu jouer?" which translates to "How will you play?" making the model inclined to translate the mentioned English sentence to "How will you?" or similar to these would be treated a mistake by a model. The model greedily outputs the most likely word at every step, it ends up with a suboptimal translation.

Beam search is a technique that gives model a chance to go back and fix mistakes it made earlier. It keeps track of a short list of the *k* most promising sentences (say, the top three), and at each decoder step it tries to extend them by one word, keeping only the *k* most likely sentences. The parameter *k* is called the *beam width*.

For example, a model needs to translate sentence “Comment vas-tu?” using beam search with a beam width of 3. At the first decoder step, the model will output an estimated probability for each possible word. Suppose the top three words are “How” (75% estimated probability), “What” (3%), and “You” (1%). Next, three copies of our model are created and used to find the next word for each sentence. Each model will output one estimated probability per word in the vocabulary. The first model will try to find the next word in the sentence “How,” and perhaps it will output different conditional probabilities for each predicted word, given that the sentence starts with “How.”. These are actually conditional probabilities, Similarly, the second model will try to complete the sentence “What”; it might output a different conditional probability for the word “are,” and so on. Assuming the vocabulary has 10,000 words, each model will output 10,000 probabilities.

Next, it computes the probabilities of each of the 30,000 two-word sentences that these models considered (3 × 10,000). This is done by multiplying the estimated conditional probability of each word by the estimated probability of the sentence it completes. For example, the estimated probability of the sentence “How” was 75%, while the estimated conditional probability of the word “will” (given that the first word is “How”) was 36%, so the estimated probability of the sentence “How will” is 75% × 36% = 27%. After computing the probabilities of all 30,000 two-word sentences, only the top 3 are kept. Perhaps they all start with the word “How”: “How will” (27%), “How are” (24%), and “How do” (12%). The sentence “How will” wins, but “How are” is not yet eliminated.

The same process is repeated. Three models are used to predict the next word in each of these three sentences, and the probabilities of all 30,000 three-word sentences are computed. Perhaps the top three will be “How are you” (10%), “How do you” (8%), and “How will you” (2%). At the next step the model may get “How do you do” (7%), “How are you <eos>” (6%), and “How are you doing” (3%). It is to be noted that “How will” was eliminated, and three perfectly reasonable translations are available. In this way, Encoder–Decoder model’s performance can be boosted simply by using it more wisely without any extra training.

Attention Mechanisms

With pretrained word embeddings, encoder-decoder model can get good translation performance for short sentences. Unfortunately, this model will be bad at translating long sentences. The problem comes from the limited short-term memory of RNNs. *Attention mechanisms* are the game-changing innovation that addressed this problem. It allows the decoder to focus on the appropriate words (as encoded by the encoder) at each time step.

Considering the example of translating English sentence “I drink milk” to French “Je bois du lait”, the representation of the word “milk” (along with all the other words) needs to be carried over many steps before it is actually used. Attention mechanism is a technique that allows the decoder to focus on the appropriate words (as encoded by the encoder) at each time step. For example, at the time step where the decoder needs to output the word “lait,” it will focus its attention on the word “milk.” This means that the path from an input word to its translation is now much shorter, so the short-term memory limitations of RNNs have much less impact. Attention mechanisms

revolutionized neural machine translation (and NLP in general), allowing a significant improvement in the state of the art, especially for long sentences (over 30 words).

The below figure shows slightly simplified version of this model's architecture. Encoder and the decoder are on the left. Instead of just sending the encoder's final hidden state to the decoder , all of its outputs are sent to the decoder. At each time step, the decoder's memory cell computes a weighted sum of all these encoder outputs: this determines which words it will focus on at this step. The weight $\alpha_{(t,i)}$ is the weight of the i^{th} encoder output at the t^{th} decoder time step. For example, if the weight $\alpha_{(3,2)}$ is much larger than the weights $\alpha_{(3,0)}$ and $\alpha_{(3,1)}$, then the decoder will pay much more attention to word number 2 ("milk") than to the other two words, at least at this time step. The rest of the decoder works just like earlier: at each time step the memory cell receives the inputs, plus the hidden state from the previous time step, and finally (although it is not represented in the diagram) it receives the target word from the previous time step (or at inference time, the output from the previous time step).

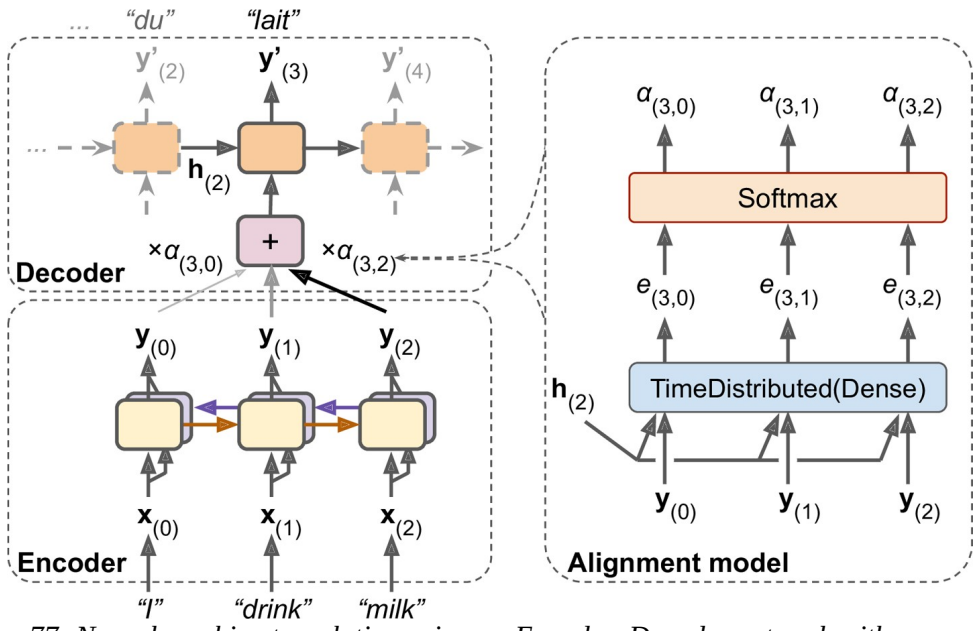


Figure 77: Neural machine translation using an Encoder–Decoder network with an attention model

The $\alpha_{(t,i)}$ weights are generated by a type of small neural network called an alignment model (or an attention layer), which is trained jointly with the rest of the Encoder–Decoder model. This alignment model is illustrated on the right-hand side of above figure. It starts with a time-distributed Dense layer with a single neuron, which receives as input all the encoder outputs, concatenated with the decoder's previous hidden state (e.g., $h_{(2)}$). This layer outputs a score (or energy) for each encoder output (e.g., $e_{(3,2)}$): this score measures how well each output is aligned with the decoder's previous hidden state. Finally, all the scores go through a softmax layer to get a final weight for each encoder output (e.g., $\alpha_{(3,2)}$). All the weights for a given decoder time step add up to 1 (since the softmax layer is not time-distributed).

Visual Attention

Attention mechanisms are now used for a variety of purposes. One of their first applications beyond neural machine translation was in generating image captions using visual attention. A convolutional neural network first processes the image and outputs some feature maps, then a decoder RNN equipped with an attention mechanism generates the caption, one word at a time. At each decoder time step (each word), the decoder uses the attention model to focus on just the right part of the image. For example, if the model generates the correct caption “A woman is throwing a frisbee in a park,”, then it can be seen that frisbee was the part of the input image the decoder focused its attention on when it was about to output the word “frisbee”.

The Transformer Architecture

Transformer is an architecture that significantly improved the neural machine translation performance without using any recurrent or convolutional layers, just by using attention mechanisms (plus embedding layers, dense layers, normalization layers, and a few other bits and pieces). This architecture is also much faster to train and easier to parallelize.

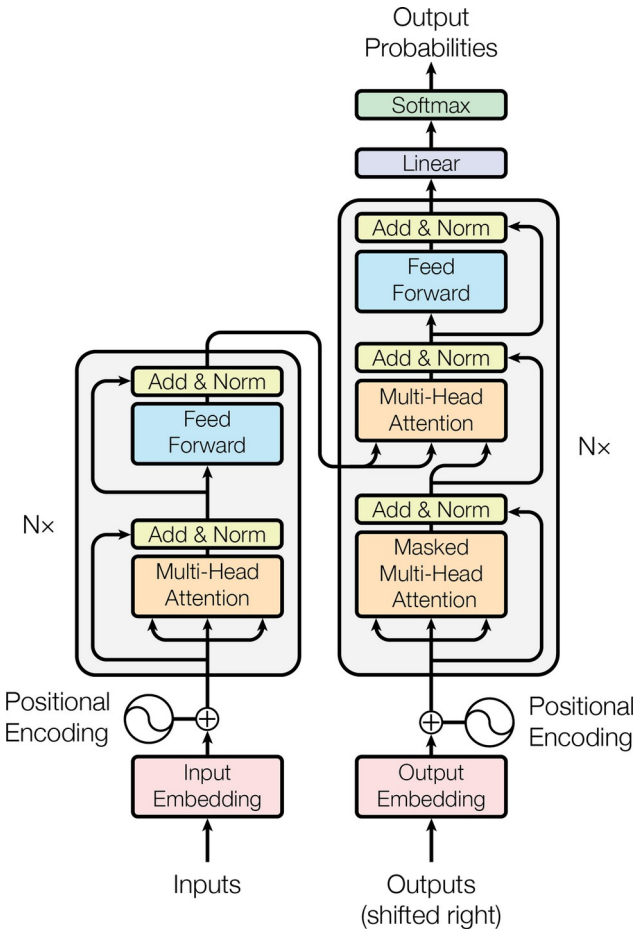


Figure 78: The Transformer architecture

The architecture is described below.

- The left-hand part is the encoder. It takes as input a batch of sentences represented as sequences of word IDs (the input shape is [batch size, max input sentence length]), and it encodes each word into a 512-dimensional representation (so the encoder's output shape is [batch size, max input sentence length, 512]). Note that the top part of the encoder is stacked *N* times.
- The right-hand part is the decoder. During training, it takes the target sentence as input (also represented as a sequence of word IDs), shifted one time step to the right (i.e., a start-of-sequence token is inserted at the beginning). It also receives the outputs of the encoder (i.e., the arrows coming from the left side). Note that the top part of the decoder is also stacked N times, and the encoder stack's final outputs are fed to the decoder at each of these N levels. The decoder outputs a probability for each possible next word, at each time step (its output shape is [batch size, max output sentence length, vocabulary length]).
- During inference, the decoder cannot be fed targets, so the previously output words are fed (starting with a start-of-sequence token). So the model needs to be called repeatedly, predicting one more word at every round (which is fed to the decoder at the next round, until the end-of-sequence token is output).
- There are two embedding layers, $5 \times N$ skip connections, each of them followed by a layer normalization layer, $2 \times N$ "Feed Forward" modules that are composed of two dense layers each (the first one using the ReLU activation function, the second with no activation function), and finally the output layer is a dense layer using the softmax activation function. All of these layers are time-distributed, so each word is treated independently of all the others.
    - Encoder's *Multi-Head Attention* layer – the most important layer in the transformer architecture, encodes each word's relationship with every other word in the same sentence, paying more attention to the most relevant ones. For example, the output of this layer for the word "Queen" in the sentence "They welcomed the Queen of the United Kingdom" will depend on all the words in the sentence, but it will probably pay more attention to the words "United" and "Kingdom" than to the words "They" or "welcomed." This attention mechanism is called self-attention (the sentence is paying attention to itself). The decoder's *Masked Multi-Head Attention* layer does the same thing, but each word is only allowed to attend to words located before it. Finally, the decoder's upper Multi-Head Attention layer is where the decoder pays attention to the words in the input sentence. For example, the decoder will probably pay close attention to the word "Queen" in the input sentence when it is about to output this word's translation.
    - The positional embeddings are simply dense vectors (much like word embeddings) that represent the position of a word in the sentence. The n^{th} positional embedding is added to the word embedding of the n^{th} word in each sentence. This gives the model access to each word's position, which is needed because the Multi-Head Attention layers do not consider the order or the position of the words; they only look at their relationships. Since all the other layers are time-distributed, they have no way of knowing the position of each word (either relative or absolute). Obviously, the relative and absolute word

positions are important, so we need to give this information to the Transformer somehow, and positional embeddings are a good way to do this.

Exercises

1) Explain with example the process of generating text using Char-RNN.
2) Explain with example the difference between a stateful RNN and a stateless RNN.
3) Explain with example how RNN can be used for sentiment analysis. Also, show how using pretrained embeddings can improve performance if training dataset is not large.
4) Show with example how encoder-decoder network can be used for neural machine translation. How can you deal with variable-length input sequences and variable-length output sequences?
5) When would you need to use sampled softmax in encoder-decoder network especially during neural machine translation.
6) Show how encoder-decoder model performance can be boosted using beam search.
7) What is an attention mechanism? How does it help improving performance in encoder-decoder model?
8) Explain Transformer architecture. What is the most important layer in the Transformer architecture? What is its purpose?

MODULE 5

Autoencoders

<PLACEHOLDER FOR CONTENT>

Exercises

Refer second edition of first textbook to relate hint provided against each exercises below. Page numbers mentioned in the hints refer soft copy of the book, not the hard copy.

1. What are the main tasks that autoencoders are used for and how does an autoencoder work

   [*Hint: First paragraph in page 567 and first bullet point in page 568*]

2. If an autoencoder perfectly reconstructs the inputs, is it necessarily a good autoencoder? How can you evaluate the performance of an autoencoder?

   [*Hint: Last paragraph in page 569, figure 17-1 in page 570 followed by two paragraphs*]

3. What are undercomplete autoencoders? Show performing PCA with an undercomplete linear autoencoder.

   [*Hint: From second paragraph in page 570 till before section "Stacked Autoencoders". Include figure 17-2*]

4. What are overcomplete autoencoders? What about the main risk of an overcomplete autoencoder?

   [*Hint: Adding more layers helps the autoencoder learn more complex codings. Care must be taken not to make the autoencoder too powerful. If an encoder is made so powerful that it just learns to map each input to a single arbitrary number (and the decoder learns the reverse mapping), obviously such an autoencoder will reconstruct the training data perfectly, but it will not have learned any useful data representation in the process and it is unlikely to generalize well to new instances.*]

5. Explain stacked autoencoders and write code-snippet to implement it considering any suitable dataset.

   [*Hint: From section "Stacked Autoencoder in page 572 before subsection "Visualizing the Fashion MNIST Dataset" in page 574. Include figure 17-3.*]

6. Suppose you want to train a classifier, and you have plenty of unlabeled training data but only a few thousand labeled instances. How can autoencoders help? How would you proceed?

   [*Hint: From section "Unsupervised Pretraining Using Stacked Autoencoders" in page 576 before subsection "Tying Weights" in page 577. Include figure 17-6.*]

7. How do you tie weights in a stacked autoencoder? What is the point of doing so?

   [*Hint: Refer section "Trying Weights on page number 577.*]

8. What is a generative model? Can you name a type of generative autoencoder?

   [*Hint: A generative model is a model capable of randomly generating outputs that resemble the training instances. For example, once trained successfully on the MNIST dataset, a generative model can be used to randomly generate realistic images of digits. The output distribution is typically similar to the training data. For example, since MNIST contains many images of each digit, the generative model would output roughly the same number of images of each digit. Some generative models can be parametrized—for example, to generate only some kinds of outputs. An example of a generative autoencoder is the variational autoencoder.*]

9. Explain variational autoencoder and it working.

   [*Hint: From section "Variational Autoencoders" in page 586 through first paragraph in page 587. Include diagram 17-12.*]

10. Explain Generative Adversarial Networks?

    [*Hint: From section "Generative Adversarial Networks" in page 592 though second bullet point in page 593.*]

11. What are the main difficulties when training GANs?

    [*Hint: From section "The Difficulties of Training GANs" in page 596 through the first three paragraphs in page 579.*]

Reinforcement Learning

<PLACEHOLDER FOR CONTENT>

Exercises

Refer second edition of first textbook to relate hint provided against each exercises below. Page numbers mentioned in the hints refer soft copy of the book, not the hard copy.

1. How would you define Reinforcement Learning? How is it different from regular supervised or unsupervised learning?

   [*Hint: From section "Learning to Optimize Rewards" till the end of section "Policy Search". Include figure 18-3.*]

2. What is the credit assignment problem? When does it occur? How can you alleviate it?

   [*Hint: First two paragraph in section "Evaluating Actions: The Credit Assignment Problem" in page 619. Include figure 18-6.*]

3. Explain Q-Learning.

   [*Hint: From section "Q-Learning" in page 630 till before section "Implementing Deep Q-Learnng" in page 634.*]

4. Examine problem associated with Q-Learning with respect to medium to large Markov Decision Process with many states and actions, and analyze two common approaches e.g. Approximate Q-Learning and Deep Q-Learning in approximating Q-Values.

   [*Hint: From section "Approximate Q-Learning and Deep Q-Learning" in page 633 till before section "Implementing Deep Q-Learnng" in page 634.*]

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