Neural Networks & Deep Learning [18AI81]

Teaching Material

Version 0.7

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.

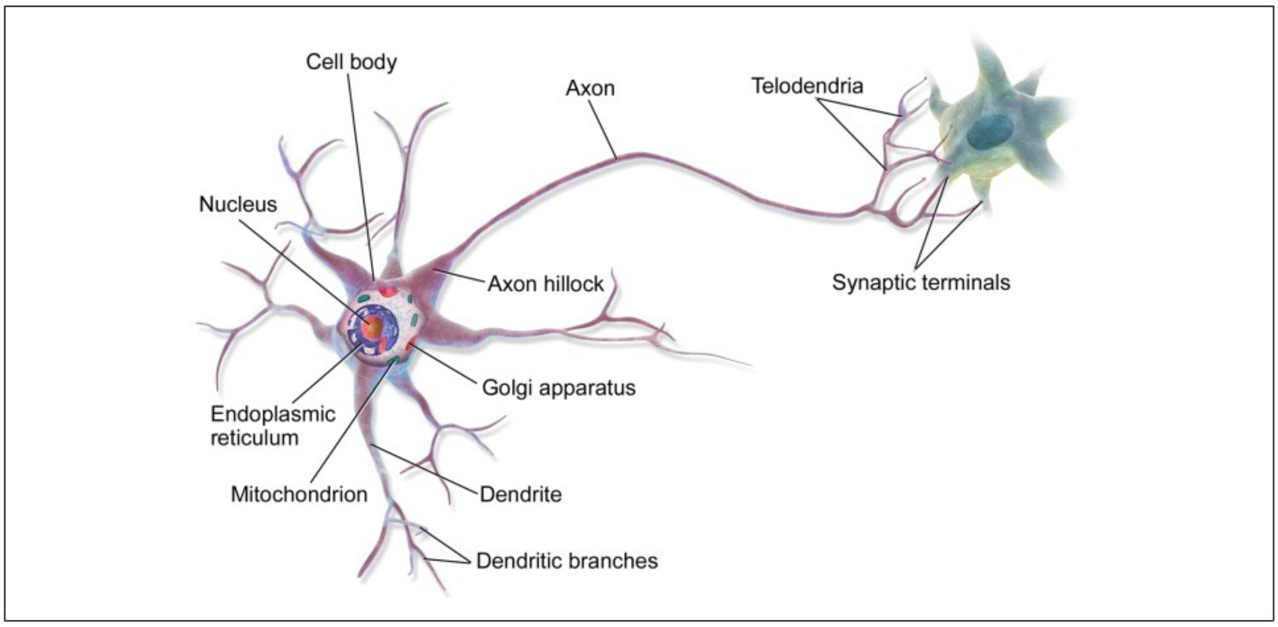


Figure 1: A biological neuron

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.

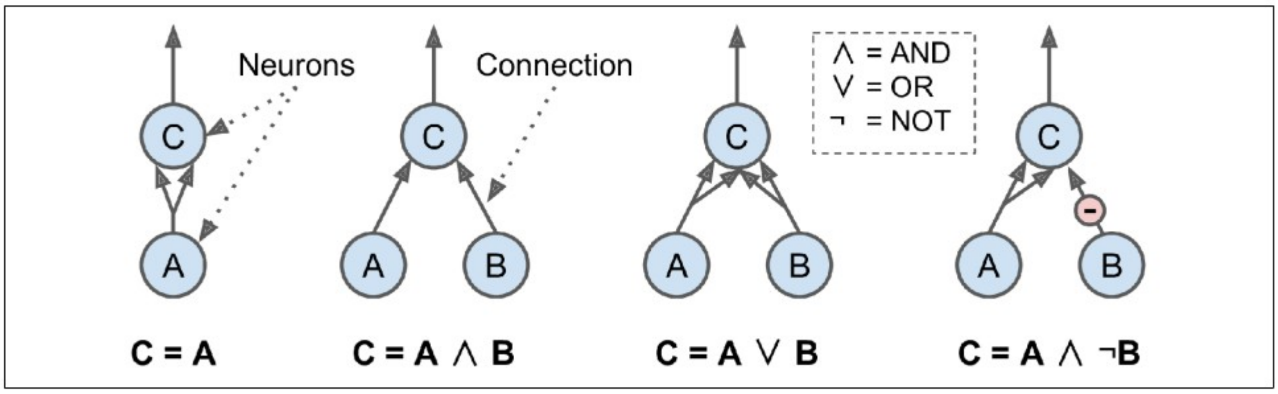
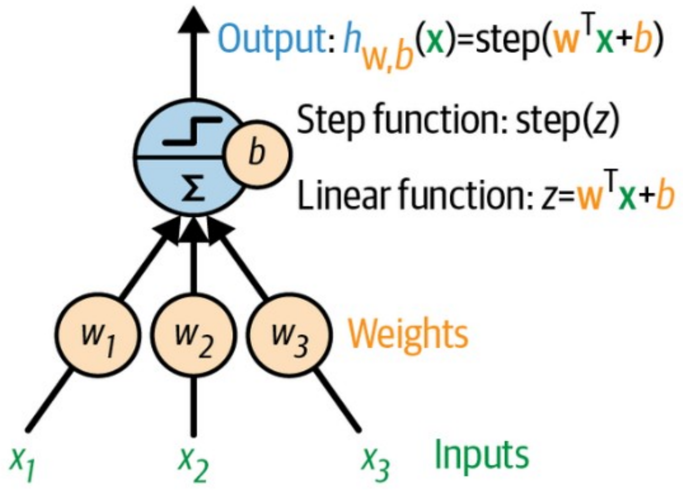


Figure 2: ANNs performing simple logical computations

- 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: $z = w_1 x_1 + w_2 x_2 + \cdots + w_n x_n + b = \mathbf{w}^T \mathbf{x} + b$. Then applies a step function to that sums and outputs the result: $h_{\mathbf{w}}(\mathbf{x}) = step(z)$. 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 **w** and bias term *b*.



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

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

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

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

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.

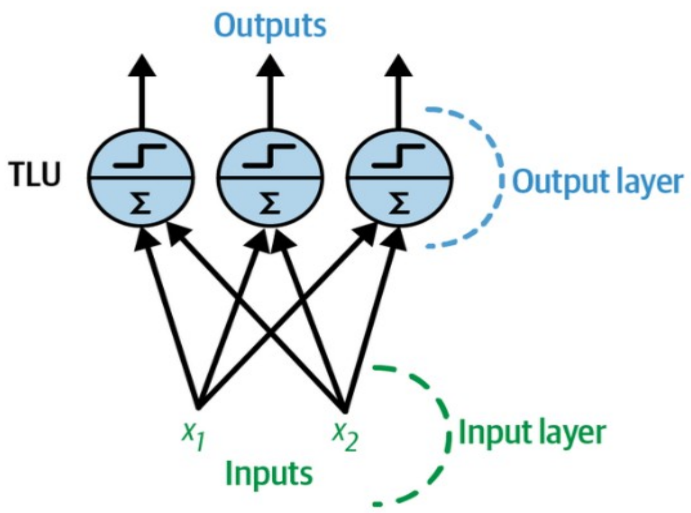


Figure 4: Architecture of a perceptron with two inputs and three output neurons

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.

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

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.

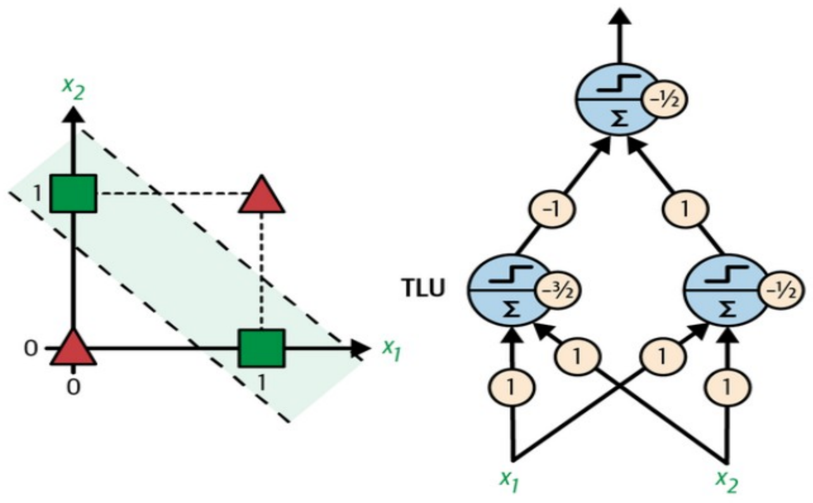


Figure 5: solving XOR function using an MLP

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).

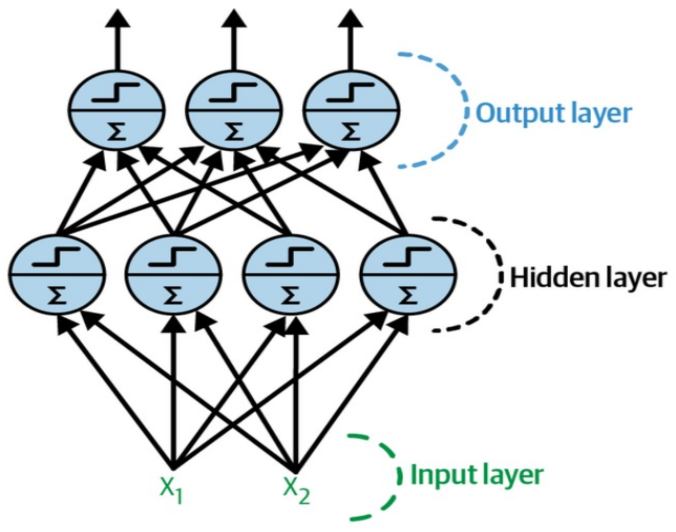


Figure 6: Architecture of a multilayer perceptron with two inputs, one hidden layer of four neurons, and three output neurons

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:

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

Equation 4: Tanh activation function

Just like the sigmoid function, this activation function is S-shaped, continuous, and differentiable, but its output value ranges from –1 to 1 (instead of 0 to 1 in the case of the sigmoid function). That range tends to make each layer’s output more or less centered around 0 at the beginning of training, which often helps speed up convergence.

The rectified linear unit function:

$$ReLU(z) = max(0, z)$$

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.

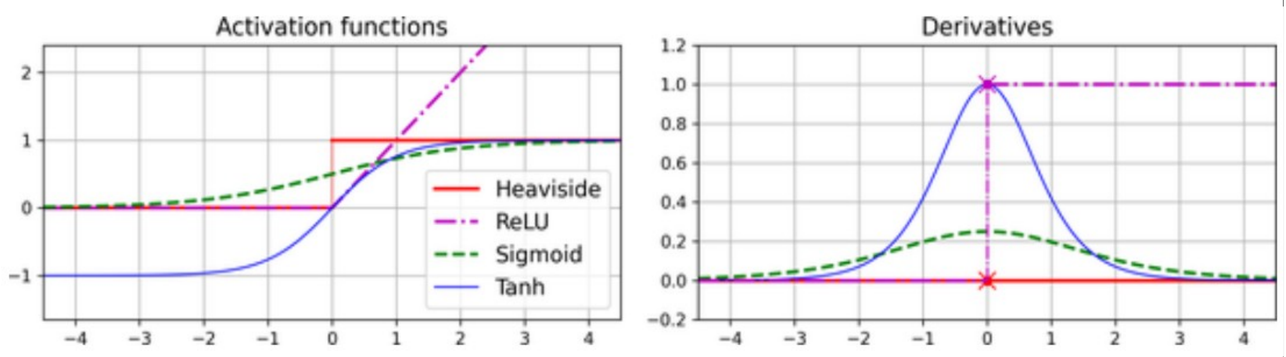


Figure 7: Popular activation functions and their derivatives

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.

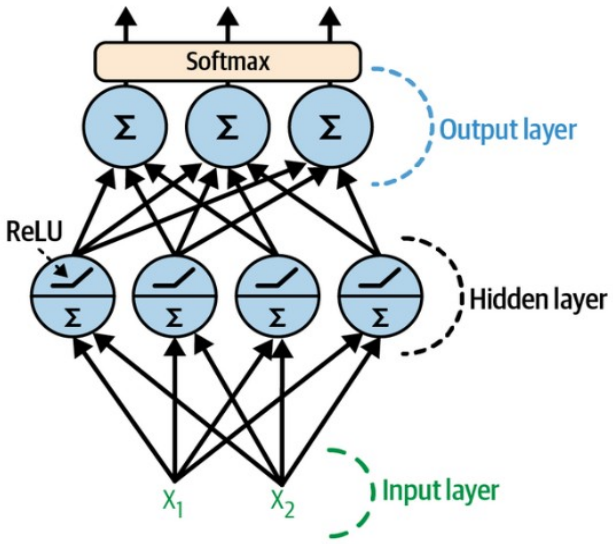


Figure 8: A MLP (with ReLU and softmax) for classification

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"]
```

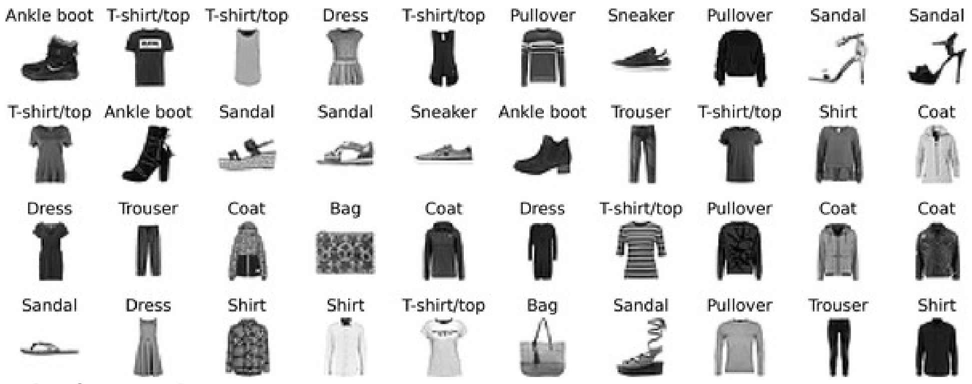


Figure 9: Samples from Fashion MNIST

Creating the model using the sequential API

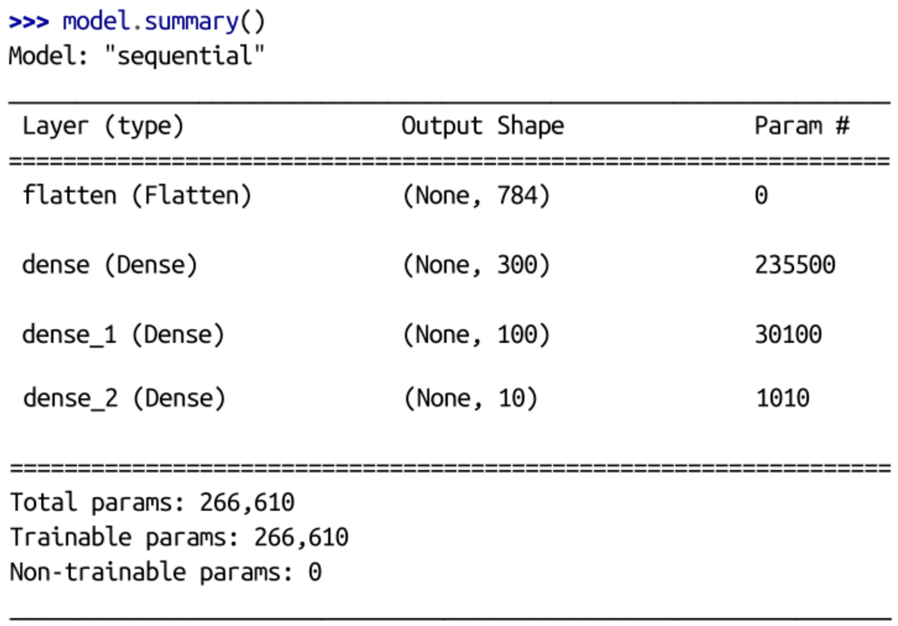
The below code builds a classification MLP with two hidden layers:

```
# 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.] 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()
```

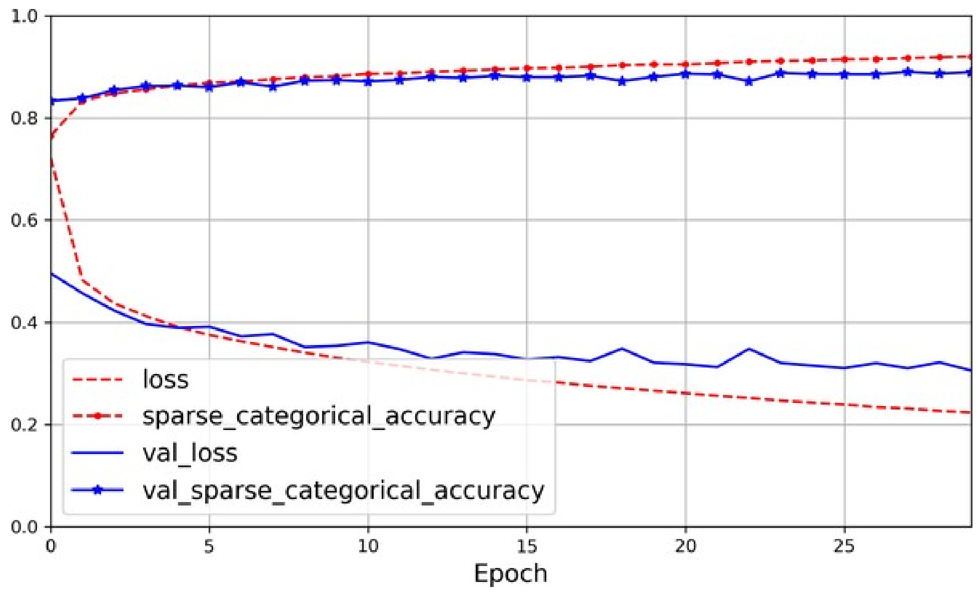


Figure 10: Learning curves: the mean loss and accuracy measured over each epoch, both for training and validation data

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).

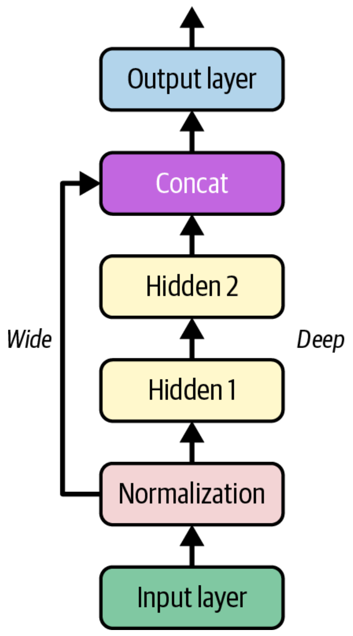


Figure 11: Wide & Deep neural network

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.

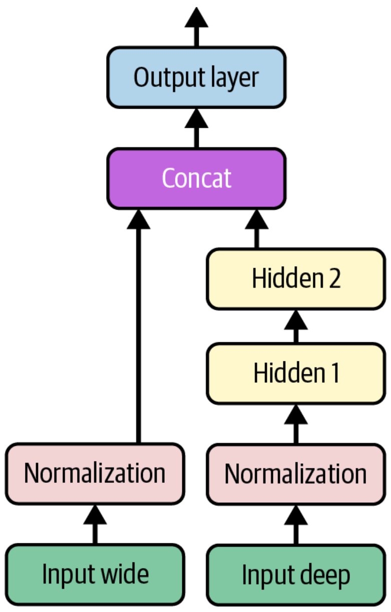


Figure 12: Handling multiple inputs

```
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.

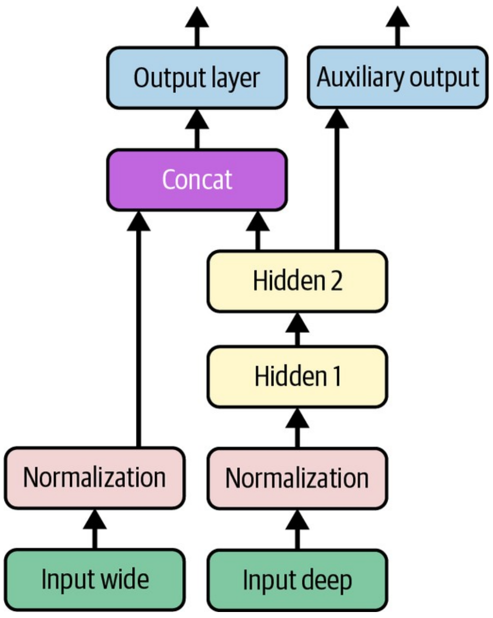


Figure 13: Handling multiple outputs

```
[...] # 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_test_begin()`
	`on_test_end()`
		`on_predict_begin()`
		`on_predict_end()`

`on_epoch_begin()`
`on_epoch_end()`
`on_batch_begin()`
`on_batch_end()`
	`on_test_batch_begin()`
	`on_test_batch_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.

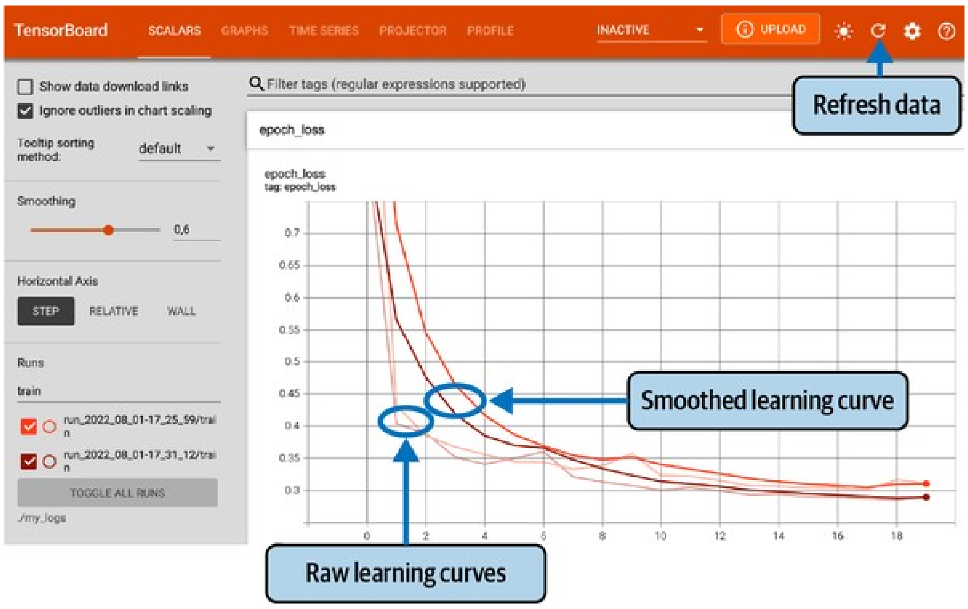


Figure 15: Visualizing learning curves with TensorBoard

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 $28 \times 28 = 784$ inputs and 10 output neurons.

Pyramid Approach: As for the hidden layers, it used to be common to size them to form a pyramid, with fewer and fewer neurons at each layer because many low-level features can coalesce into far fewer high-level features. A typical neural network for MNIST might have 3 hidden layers, the first with 300 neurons, the second with 200, and the third with 100. However, this practice has been largely abandoned because it seems that using the same number of neurons in all hidden layers performs just as well in most cases, or even better; plus, there is only one hyperparameter to tune, instead of one per layer. Depending on the dataset, it can sometimes help to make the first hidden layer bigger than the others.

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

Learning Rate, Batch Size, and Other Hyperparameters

Learning rate

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

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

Optimizer

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

Several advanced optimizers are explained in section Faster Optimizers in chapter *Training Deep Neural Networks*.

Batch size

The batch size can have a significant impact on the model's performance and training time. The main benefit of using large batch sizes is that hardware accelerators like GPUs can process them efficiently. Therefore, many researchers and practitioners recommend using the largest batch size that can fit in GPU RAM. But in practice, large batch sizes often lead to training instabilities, especially at the beginning of training, and the resulting model may not generalize as well as a model trained with a small batch size. In general, using small batches (of 32, for example) is preferable because small batches led to better models in less training time. Research also showed that it was possible to use very large batch sizes (up to 8,192) along with various techniques such as learning rate warming up (i.e., starting training with a small learning rate, then ramping it up) and to obtain very short training times, without any generalization gap. So, one strategy is to try to using a large batch size, with learning rate warm-up, and if training is unstable or the final performance is disappointing, then try using a small batch size instead.

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

Activation function

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

Number of iterations

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

Exercises

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

1. Draw an ANN using the original artificial neurons (McCulloch-Pitts artificial neuron) that computes $A \oplus B$ (where $\oplus$ represents the XOR operation). Hint: $A \oplus B = (A \wedge \neg B) \vee (\neg A \wedge B)$.
2. Why is it generally preferable to use a logistic regression classifier rather than a classic perceptron (i.e., a single layer of threshold logic units trained using the perceptron training algorithm)? How can you tweak a perceptron to make it equivalent to a logistic regression classifier?
3. Why was the sigmoid activation function a key ingredient in training the first MLPs?
4. Write code snippet to build an MLP model to predict house price using California housing dataset and justify the assumptions taken. Consider using Scikit Learn implementation of MLPs.

5. Write code snippet to build an image classifier using Keras Sequential API using Fashion MNIST dataset and to evaluate the model, and also explain rationale behind every assumptions.
6. Name three popular activation functions. Can you draw them?
7. Suppose you have an MLP composed of one input layer with 10 pass-through neurons, followed by one hidden layer with 50 artificial neurons, and finally one output layer with 3 artificial neurons. All artificial neurons use the ReLU activation function.
   a) What is the shape of the input matrix **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.

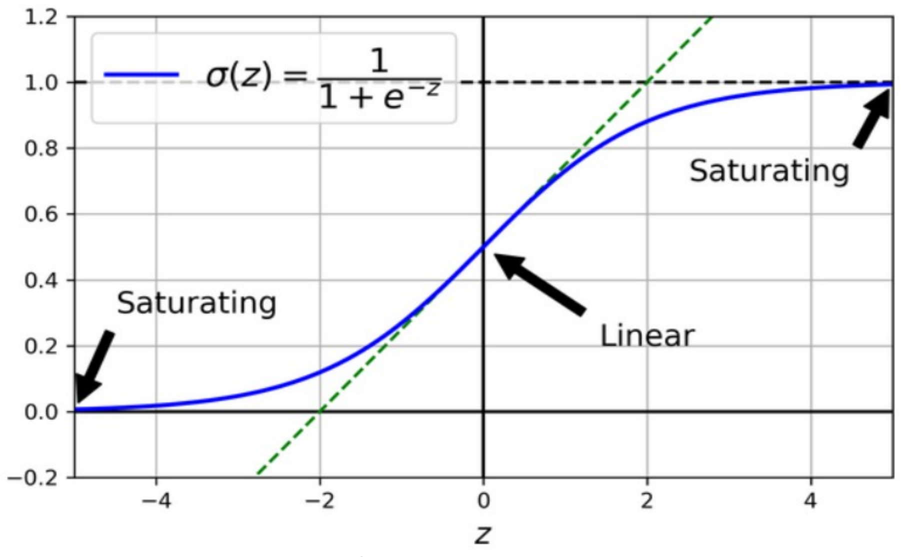


Figure 16: Sigmoid activation function saturation

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*.

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

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

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 $\alpha = 0.01$. 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.

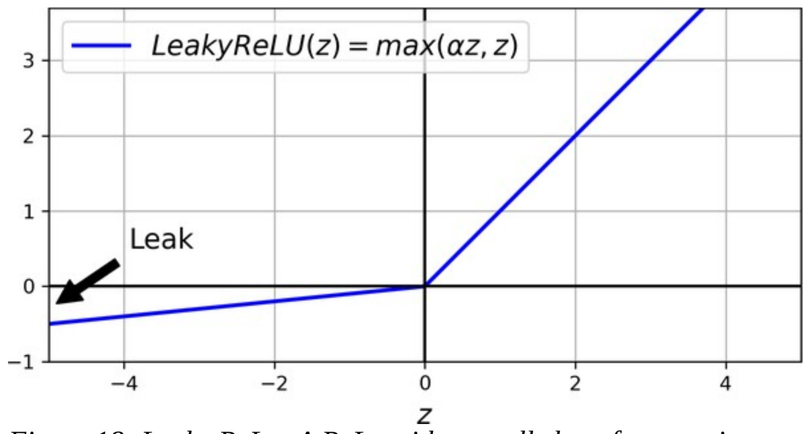


Figure 18: Leaky ReLu: A ReLu with a small slope for negative values

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.

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

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 $z = 0$.

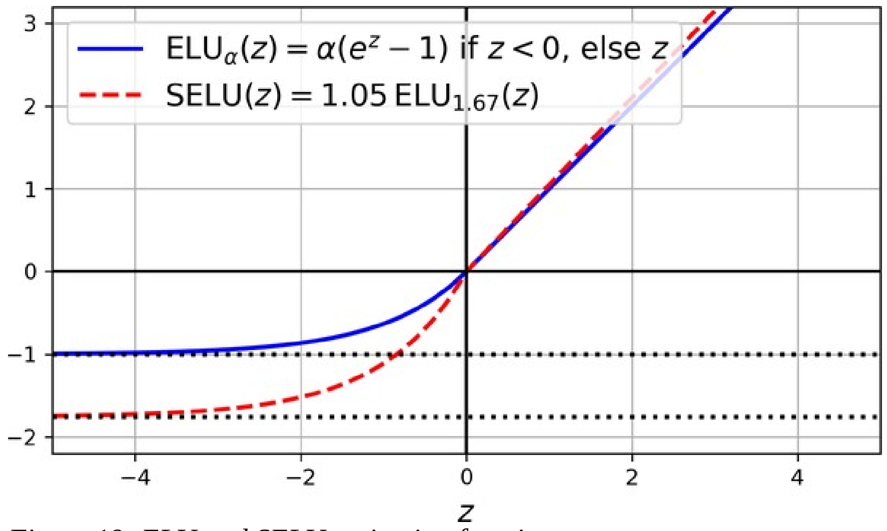


Figure 19: ELU and SELU activation functions

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.

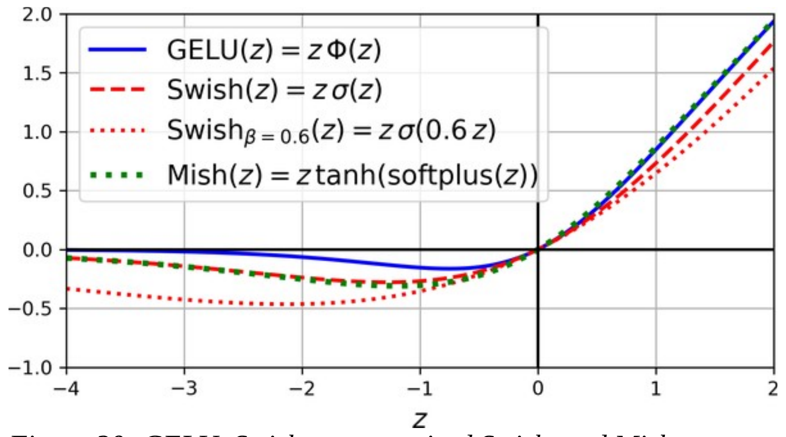


Figure 20: GELU, Swish, parametrized Swish, and Mish activation functions

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

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.
- $\hat{\mathbf{x}}^{(i)}$ 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 $W' = \boldsymbol{\gamma} \otimes \boldsymbol{W} / \boldsymbol{\sigma}$ 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: γ, β, μ, and σ (for example, the first BN layer adds 3,136 parameters, which is 4 × 784). Parameters, μ and σ are the moving averages and are non-trainable where parameters γ and β 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.

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.

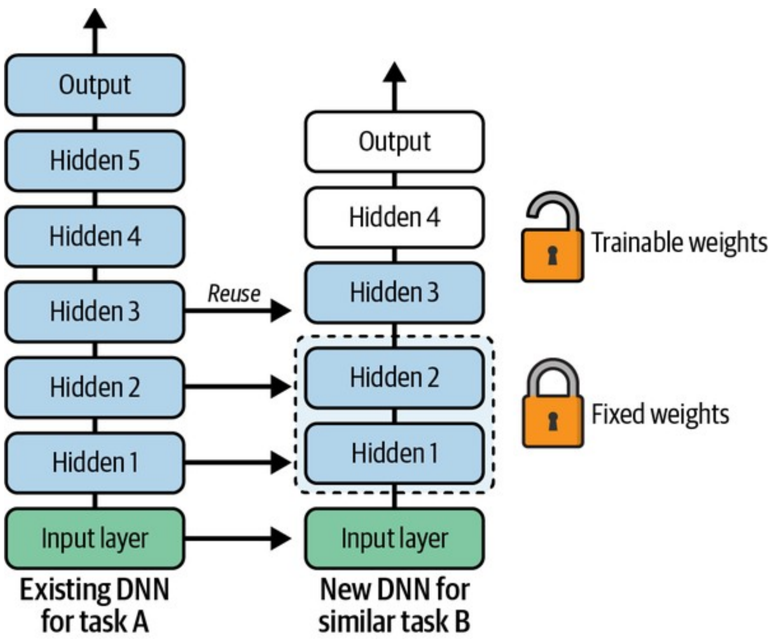


Figure 21: Reusing pretrained layers

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.

While 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 with the equation is $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta\nabla_\theta J(\boldsymbol{\theta})$ and without caring about what the earlier gradients were, momentum optimization takes a great deal about what previous gradients were: at each iteration, it subtracts the local gradient from the momentum vector m (multiplied by the learning rate η), and it updates the weights by adding this momentum vector (refer equation below).

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

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 – β) (ignoring the sign). For example, if β = 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).

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

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 $\boldsymbol{\theta} + \beta\mathbf{m}$.

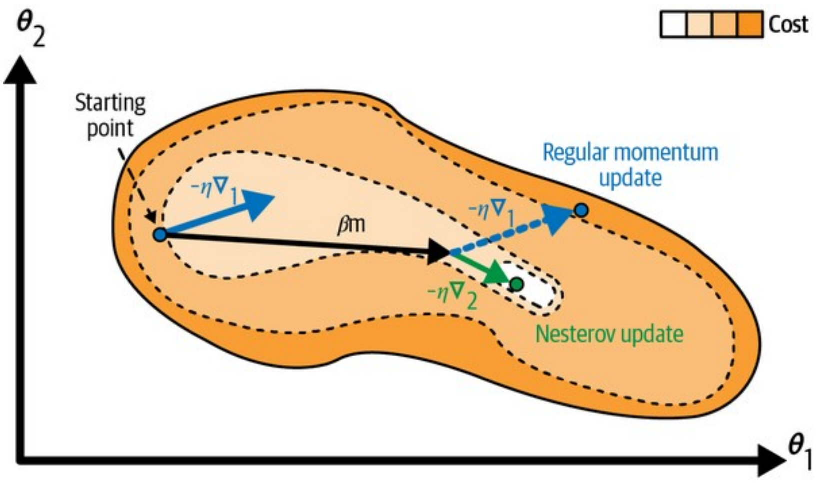


Figure 22: Regular versus Nesterov momentum optimization: the former applies the gradients

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).

1. $s \leftarrow s + \nabla_\theta J(\theta) \otimes \nabla_\theta J(\theta)$
2. $\theta \leftarrow \theta - \eta \nabla_\theta J(\theta) \oslash \sqrt{s + \varepsilon}$

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 $\sqrt{s + \varepsilon}$ (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.

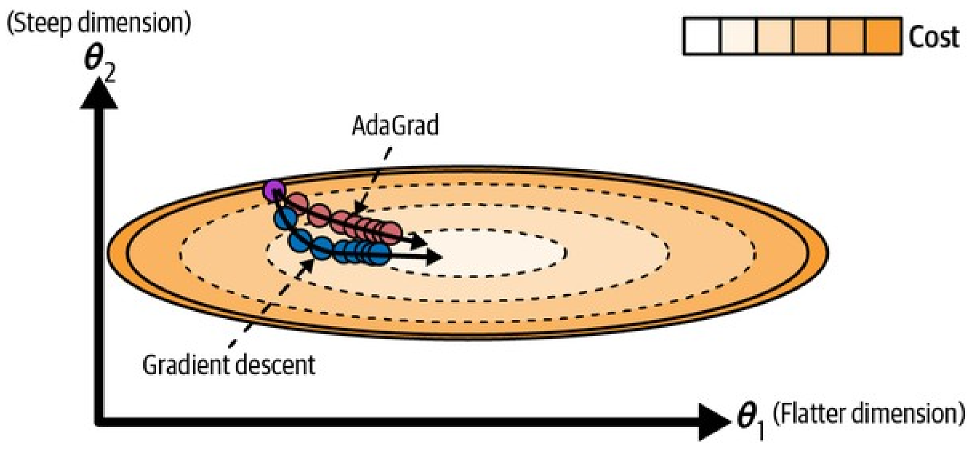


Figure 23: AdaGrad versus gradient descent: the former can correct its direction earlier to point to the optimum

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).

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

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

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

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

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 $s \leftarrow max(\beta_2 s, \left|\nabla_\theta J(\theta)\right|)$, 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.

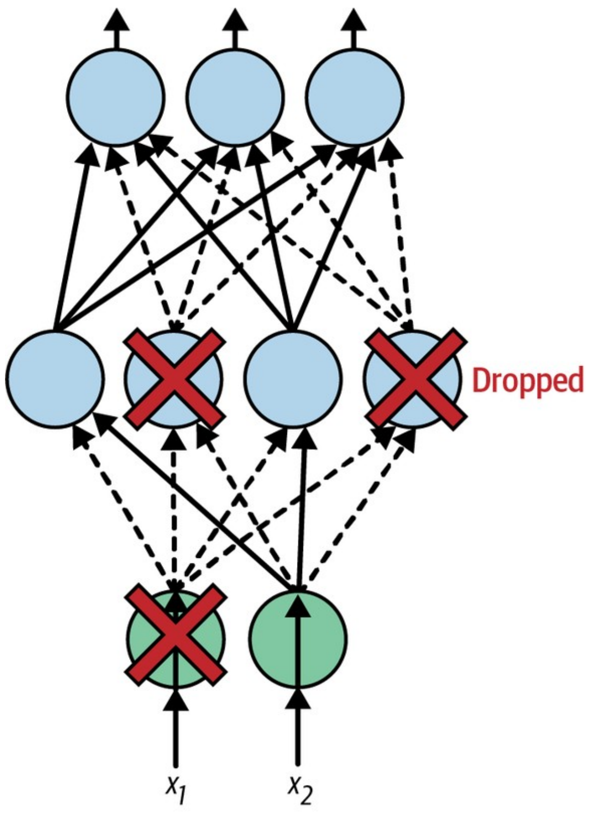


Figure 24: Dropout regularization: at each training iteration, a random subset of all neurons in one or more layers, except the output layer, are dropped out; these neurons output 0 at this iteration (represented by the dashed arrows)

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?

MODULE 3

<PLACEHOLDER>

MODULE 4

<PLACEHOLDER>

MODULE 5

<PLACEHOLDER>

REFERENCES

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