

Tuning a Feedforward network

Building a feedforward neural network is easy. Tuning a neural net is not so easy. You want to avoid just trying random stuff.

In this assignment we'll create some infrastructure to make tuning more systematic, and then use it. You'll also get exposed to some other useful things:

- how Keras models can be returned by functions
- how feedforward nets can handle large inputs, like images
- how the results of successive runs of training can give quite different results

We'll build a multi-class classifier for the famous CIFAR-10 image data set, which consists of 32 x 32 pixel color images of objects in 10 categories.

Instructions:

- Please start by reading the notebook carefully.
- Then provide answers to the numbered problems 1-5.
- Only modify cells as instructed.
- It may not be possible for you to restart and run all before submitting.

Hints:

- The GPU option for Colab's runtime type (see Runtime menu) is a lot faster than the default CPU.
- I created this notebook on Colab because many of you use Colab. However, I ran out of my free allocation of resources, and ended up paying for ColabPro (\$10/month, paid monthly). You may want to do your work in Jupyter on your own machine.
- Tuning takes time. You may want to run grid searches overnight. Get started early enough to make this possible.

Read and preprocess the image data

Read the data

As mentioned, this is a famous data set, and it is a built-in Keras data set. The images in this data set are tiny, only 32 by 32 pixels in size. They belong to 10 classes, 0 for airplane, 1 for automobile, etc.

See [the Keras CIFAR-10 page](#).

```
(50000, 32, 32, 3)
(50000, 1)
(10000, 32, 32, 3)
(10000, 1)
```

There are 50K training images and 10K test images.

The shape of the data is typical of color image data:

```
X_train[image, row, col, color]
```

So `X_train[0]` is the first image, `X_train[1]` is the second image, etc.

In the row position, values can range from 0-31, and the same for the col position.

In the color position, we have red/green/blue (RGB) color "channels", with a value of 0 for red, 1 for green, and 2 for blue.

The label values are 0 through 9. They are stored in a 2D NumPy array instead of a 1D array.

```
[[6]
 [9]
 [9]
 [4]
 [1]
 [1]
 [2]
 [7]
 [8]
 [3]]
```

Some random images



Get a subset of the data

We won't use the full data set because it would slow training down.

Flatten and scale the data

A single input will be a long array of values. The length of the array is $32 * 32 * 3$.

Squeeze the dimensions of `y_train`, `y_test`

The label data will be stored in 1D arrays, not 2D arrays.

```
(25000, 3072)
(25000,)
(10000, 3072)
(10000,)
```

```
10
```

A feedforward neural network

You may know that *convolutional* neural nets are commonly used for image classification. But we'll use a simple feedforward net here, and will be able to compare performance with a convnet later.

Create a model with 2 hidden layers plus an output layer. There will be an output for each class.

Problem 1. Create a Keras model for CIFAR-10 image classification with 2 hidden layers, each containing 10 neurons. Use `num_classes` when defining the number of outputs. Use RELU activation in the hidden layers. You'll need to use softmax activation in the output layer.

```
2024-10-10 10:25:06.480913: I metal_plugin/src/device/metal_device.cc:1154] Metal device set to: Apple M2 Max
2024-10-10 10:25:06.480964: I metal_plugin/src/device/metal_device.cc:296] systemMemory: 32.00 GB
2024-10-10 10:25:06.480983: I metal_plugin/src/device/metal_device.cc:313] maxCacheSize: 10.67 GB
2024-10-10 10:25:06.480999: I tensorflow/core/common_runtime/pluggable_device/pluggable_device_factory.cc:305] Could not identify NUMA node of platform GPU ID 0, defaulting to 0. Your kernel may not have been built with NUMA support.
2024-10-10 10:25:06.481018: I tensorflow/core/common_runtime/pluggable_device/pluggable_device_factory.cc:271] Created TensorFlow device (/job:localhost/replica:0/task:0/device:GPU:0 with 0 MB memory) -> physical PluggableDevice (device: 0, name: METAL, pci bus id: <undefined>)
```

The model has a lot of parameters in its first hidden layer, because each neuron in that layer has many inputs.

Some people recommend that the number of parameters in each hidden layer should be approximately the same.

Model: "sequential"

Layer (type)	Output Shape	Param #
dense (Dense)	(None, 10)	30,730
dense_1 (Dense)	(None, 10)	110
dense_2 (Dense)	(None, 10)	110

Total params: 30,950 (120.90 KB)

Trainable params: 30,950 (120.90 KB)

Non-trainable params: 0 (0.00 B)











Train the model

When evaluating accuracy, remember that our baseline is 0.1, because that is the accuracy we'd achieve by guessing, given equal number of images of each of 10 classes.

Epoch 1/10

```
2024-10-10 10:25:33.004515: I tensorflow/core/grappler/optimizers/custom_graph_optimizer_registry.cc:117] Plugin optimizer for device_type GPU is enabled.
```

```


547/547  4s 6ms/step - accuracy: 0.0981 - loss: 2.3044 - val_accu
racy: 0.0977 - val_loss: 2.3027
Epoch 2/10
547/547  3s 5ms/step - accuracy: 0.1042 - loss: 2.3024 - val_accu
racy: 0.0976 - val_loss: 2.3027
Epoch 3/10
547/547  3s 5ms/step - accuracy: 0.1006 - loss: 2.3029 - val_accu
racy: 0.1056 - val_loss: 2.2876
Epoch 4/10
547/547  3s 5ms/step - accuracy: 0.1490 - loss: 2.2710 - val_accu
racy: 0.1624 - val_loss: 2.2139
Epoch 5/10
547/547  3s 6ms/step - accuracy: 0.1665 - loss: 2.1868 - val_accu
racy: 0.1652 - val_loss: 2.1620
Epoch 6/10
547/547  3s 6ms/step - accuracy: 0.1646 - loss: 2.1486 - val_accu
racy: 0.1761 - val_loss: 2.1148
Epoch 7/10
547/547  3s 6ms/step - accuracy: 0.1710 - loss: 2.1188 - val_accu
racy: 0.1712 - val_loss: 2.1192
Epoch 8/10
547/547  3s 5ms/step - accuracy: 0.1687 - loss: 2.1045 - val_accu
racy: 0.1856 - val_loss: 2.0867
Epoch 9/10
547/547  3s 5ms/step - accuracy: 0.1758 - loss: 2.0933 - val_accu
racy: 0.1780 - val_loss: 2.0803
Epoch 10/10
547/547  3s 5ms/step - accuracy: 0.1732 - loss: 2.0951 - val_accu
racy: 0.1833 - val_loss: 2.0742

```

Use the model to make predictions

Note that the output is a probability distribution over the possible target values.

```

1/1  0s 38ms/step
array([0.07295164, 0.06295176, 0.13661493, 0.12340397, 0.12583911,
       0.12652598, 0.1302514 , 0.13970731, 0.03472402, 0.04702993],
      dtype=float32)

```

Notice that the sum of the predictions is 1, and the predicted class can be found with `argmax()`.

```


1.0000001
7

```

Evaluating the model on new data

This illustrate the model method `evaluate()`, which we haven't discussed in class.

```

313/313  1s 4ms/step - accuracy: 0.1917 - loss: 2.0789
test accuracy: 0.187

```

Early stopping

Keras supports callbacks, which are basically functions that can be provided to Keras to be run during training. You can write custom callback functions, but Keras provides a few standard ones, including one for "early stopping".

The idea of early stopping is that the training process can be stopped when a certain metric stops increasing or decreasing. In this case, we want to stop training when validation loss stops lowering.

The validation loss can go up and down a little randomly, so we don't want to stop training as soon as validation loss does not decrease. We might want to "be patience" and wait a few epochs to see if the value gets lower.

```
Epoch 1/20
547/547 ██████████ 4s 7ms/step - accuracy: 0.1819 - loss: 2.0842 - val_accu
racy: 0.1785 - val_loss: 2.0833
Epoch 2/20
547/547 ██████████ 3s 5ms/step - accuracy: 0.1787 - loss: 2.0877 - val_accu
racy: 0.1819 - val_loss: 2.0702
Epoch 3/20
547/547 ██████████ 3s 5ms/step - accuracy: 0.1823 - loss: 2.0798 - val_accu
racy: 0.1883 - val_loss: 2.0819
Epoch 4/20
547/547 ██████████ 3s 5ms/step - accuracy: 0.1785 - loss: 2.0794 - val_accu
racy: 0.1825 - val_loss: 2.0890
```

Package up the model in a function

It is common to write a function to build and return a Keras model. One reason to do this is to make it easy to create customized versions of a model.

Problem 2. Write a function that will return a model with the given activation function and the given number of hidden layers and hidden layer sizes. Only the first hidden layer should provide an input shape. You can use the shape of `X_train` to determine the input shape.

A value of `layer_sizes = [10, 5]` means "2 hidden layers, with the first containing 10 neurons and the second containing 5 neurons".

Hint: your code will be simpler if you use a Keras `InputLayer`, as opposed to defining the input shape in the first dense layer.

By default, we get a model with one hidden layer of 10 neurons, and the `relu` activation function in the hidden layers.

Model: `"sequential_1"`

Layer (type)	Output Shape	Param #
dense_3 (Dense)	(None, 10)	30,730
dense_4 (Dense)	(None, 10)	110

Total params: 30,840 (120.47 KB)

Trainable params: 30,840 (120.47 KB)

Non-trainable params: 0 (0.00 B)

An example with 10 hidden layers, and ELU activation.

Model: "sequential_2"


Layer (type)	Output Shape	Param #
dense_5 (Dense)	(None, 10)	30,730
dense_6 (Dense)	(None, 5)	55
dense_7 (Dense)	(None, 10)	60

Total params: 30,845 (120.49 KB)


Trainable params: 30,845 (120.49 KB)

Non-trainable params: 0 (0.00 B)


Epoch 1/5

547/547  5s 9ms/step - accuracy: 0.1608 - loss: 2.3287 - val_accuracy: 0.2376 - val_loss: 2.0240


Epoch 2/5

547/547  4s 8ms/step - accuracy: 0.2415 - loss: 2.0278 - val_accuracy: 0.2503 - val_loss: 2.0116


Epoch 3/5

547/547  4s 8ms/step - accuracy: 0.2800 - loss: 1.9360 - val_accuracy: 0.3144 - val_loss: 1.8698

Epoch 4/5

547/547  5s 8ms/step - accuracy: 0.2941 - loss: 1.8926 - val_accuracy: 0.3272 - val_loss: 1.8437

Epoch 5/5

547/547  5s 8ms/step - accuracy: 0.3080 - loss: 1.8612 - val_accuracy: 0.3236 - val_loss: 1.8370

Here the average validation accuracy for the final two epochs is reported.

averaged validation accuracy: 0.325

Grid search

Here we define the default hyperparameter values. If we want to include more hyperparameters, this dictionary must be extended.

A function to evaluate hyperparameter settings

In searching for good hyperparameter settings, it is handy to write a function that computes how well a set of hyperparameter settings work.

This function builds a model, compiles it, trains it, and then reports on the model accuracy, using the hyperparameters in 'params'.

A function to perform grid search

With our `evaluate_params()` function, it's easy to create a function to perform hyperparameter search.

To define the parameter combinations we want to search, we will use a dictionary like this, where the keys are hyperparameter names and the values are lists of possible hyperparameter values:

```
{'act_fun': ['relu', 'elu'], 'layer_sizes': [[10], [10,5]]}
```

It's also possible to use a list of such dictionaries, like this:


```
[{'act_fun': ['relu'], 'layer_sizes': [[10], [20]]}, {'act_fun': ['elu'],  
'layer_sizes': [[20], [50]]}]
```

A grid search example


Look at the params carefully to make sure you understand how `ParameterGrid()` works.

params: {'act_fun': 'relu', 'layer_sizes': [20]}


Epoch 1/50

2188/2188  **13s** 6ms/step - accuracy: 0.1018 - loss: 2.3258 - val_accuracy: 0.1005 - val_loss: 2.3029

Epoch 2/50


2188/2188  **14s** 6ms/step - accuracy: 0.0983 - loss: 2.3032 - val_accuracy: 0.0976 - val_loss: 2.3030

Epoch 3/50


2188/2188  **10s** 4ms/step - accuracy: 0.1025 - loss: 2.3021 - val_accuracy: 0.0976 - val_loss: 2.3031

params: {'act_fun': 'relu', 'layer_sizes': [10, 10]}


Epoch 1/50

2188/2188  **13s** 6ms/step - accuracy: 0.0989 - loss: 2.3077 - val_accuracy: 0.0976 - val_loss: 2.3029

Epoch 2/50


2188/2188  **11s** 5ms/step - accuracy: 0.0960 - loss: 2.3028 - val_accuracy: 0.0976 - val_loss: 2.3031

Epoch 3/50


2188/2188  **11s** 5ms/step - accuracy: 0.1041 - loss: 2.3025 - val_accuracy: 0.0975 - val_loss: 2.3029

params: {'act_fun': 'elu', 'layer_sizes': [20]}


Epoch 1/50

2188/2188  **13s** 6ms/step - accuracy: 0.2110 - loss: 2.2431 - val_accuracy: 0.2756 - val_loss: 2.0421


Epoch 2/50

2188/2188  **12s** 6ms/step - accuracy: 0.3008 - loss: 1.9389 - val_accuracy: 0.2833 - val_loss: 1.8953


Epoch 3/50

2188/2188  **11s** 5ms/step - accuracy: 0.3351 - loss: 1.8582 - val_accuracy: 0.3371 - val_loss: 1.8874


Epoch 4/50

2188/2188  **14s** 6ms/step - accuracy: 0.3632 - loss: 1.7966 - val_accuracy: 0.3009 - val_loss: 1.9396


Epoch 5/50

2188/2188  **17s** 8ms/step - accuracy: 0.3820 - loss: 1.7490 - val_accuracy: 0.3604 - val_loss: 1.8284

Epoch 6/50

2188/2188  **14s** 6ms/step - accuracy: 0.3787 - loss: 1.7288 - val_accuracy: 0.3448 - val_loss: 1.9135


Epoch 7/50

2188/2188  **13s** 6ms/step - accuracy: 0.3938 - loss: 1.7022 - val_accuracy: 0.3681 - val_loss: 1.7996


Epoch 8/50

2188/2188  **25s** 12ms/step - accuracy: 0.4030 - loss: 1.6781 - val_accuracy: 0.3748 - val_loss: 1.7888


Epoch 9/50

2188/2188  **11s** 5ms/step - accuracy: 0.4183 - loss: 1.6464 - val_accuracy: 0.3421 - val_loss: 1.8521

Epoch 10/50













2188/2188  **11s** 5ms/step - accuracy: 0.4239 - loss: 1.6340 - val_accuracy: 0.3951 - val_loss: 1.6999

Epoch 11/50

2188/2188  **11s** 5ms/step - accuracy: 0.4191 - loss: 1.6288 - val_accuracy: 0.3708 - val_loss: 1.8138

Epoch 12/50

```

2188/2188  11s 5ms/step - accuracy: 0.4292 - loss: 1.6077 - val_a
ccuracy: 0.3940 - val_loss: 1.7238
params: {'act_fun': 'elu', 'layer_sizes': [10, 10]}
Epoch 1/50
2188/2188  15s 7ms/step - accuracy: 0.2021 - loss: 2.1798 - val_a
ccuracy: 0.2924 - val_loss: 1.9221
Epoch 2/50
2188/2188  15s 7ms/step - accuracy: 0.2993 - loss: 1.9110 - val_a
ccuracy: 0.3113 - val_loss: 1.8900
Epoch 3/50
2188/2188  15s 7ms/step - accuracy: 0.3250 - loss: 1.8593 - val_a
ccuracy: 0.3189 - val_loss: 1.8869
Epoch 4/50
2188/2188  15s 7ms/step - accuracy: 0.3385 - loss: 1.8233 - val_a
ccuracy: 0.3469 - val_loss: 1.7791
Epoch 5/50
2188/2188  15s 7ms/step - accuracy: 0.3408 - loss: 1.8062 - val_a
ccuracy: 0.2949 - val_loss: 1.9963
Epoch 6/50
2188/2188  15s 7ms/step - accuracy: 0.3584 - loss: 1.7771 - val_a
ccuracy: 0.3584 - val_loss: 1.7703
Epoch 7/50
2188/2188  15s 7ms/step - accuracy: 0.3654 - loss: 1.7432 - val_a
ccuracy: 0.3177 - val_loss: 1.8854
Epoch 8/50
2188/2188  17s 8ms/step - accuracy: 0.3633 - loss: 1.7506 - val_a
ccuracy: 0.3647 - val_loss: 1.7692
Epoch 9/50
2188/2188  15s 7ms/step - accuracy: 0.3777 - loss: 1.7245 - val_a
ccuracy: 0.3711 - val_loss: 1.7391
Epoch 10/50
2188/2188  15s 7ms/step - accuracy: 0.3765 - loss: 1.7298 - val_a
ccuracy: 0.3557 - val_loss: 1.7828
Epoch 11/50
2188/2188  15s 7ms/step - accuracy: 0.3802 - loss: 1.7110 - val_a
ccuracy: 0.3487 - val_loss: 1.7627

```







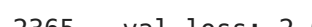






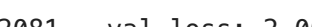



`scores_to_dataframe()` is very useful in getting a summary of the result of a grid search.

	act_fun	layer_sizes	optimizer	batch_size	0
0	relu	[20]	rmsprop	8	0.097600
1	relu	[10, 10]	rmsprop	8	0.097533
2	elu	[20]	rmsprop	8	0.382400
3	elu	[10, 10]	rmsprop	8	0.352200













What is the variation in accuracy across multiple training runs?

If you train a model multiple times, you will get a different accuracy value each time. This is super important in hyperparameter tuning, because you can mistake noise for an improvement due to a change in hyperparameter values.

Let's do a test by evaluating the same model multiple times.

```
params: {}
Epoch 1/50
2188/2188  10s 5ms/step - accuracy: 0.1001 - loss: 2.3213 - val_a
ccuracy: 0.1047 - val_loss: 2.3025
Epoch 2/50
2188/2188  10s 4ms/step - accuracy: 0.1019 - loss: 2.3029 - val_a
ccuracy: 0.1061 - val_loss: 2.3021
Epoch 3/50
2188/2188  10s 4ms/step - accuracy: 0.1042 - loss: 2.3000 - val_a
ccuracy: 0.1248 - val_loss: 2.2853
Epoch 4/50
2188/2188  9s 4ms/step - accuracy: 0.1327 - loss: 2.2751 - val_ac
curacy: 0.1723 - val_loss: 2.1831
Epoch 5/50
2188/2188  10s 4ms/step - accuracy: 0.1682 - loss: 2.1613 - val_a
ccuracy: 0.1800 - val_loss: 2.0908
Epoch 6/50
2188/2188  10s 4ms/step - accuracy: 0.1787 - loss: 2.1078 - val_a
ccuracy: 0.1928 - val_loss: 2.0620
Epoch 7/50
2188/2188  10s 4ms/step - accuracy: 0.2036 - loss: 2.0699 - val_a
ccuracy: 0.2365 - val_loss: 2.0144
Epoch 8/50
2188/2188  10s 4ms/step - accuracy: 0.2303 - loss: 2.0253 - val_a
ccuracy: 0.2355 - val_loss: 1.9767
Epoch 9/50
2188/2188  9s 4ms/step - accuracy: 0.2361 - loss: 2.0013 - val_ac
curacy: 0.2253 - val_loss: 1.9961
Epoch 10/50
2188/2188  9s 4ms/step - accuracy: 0.2476 - loss: 1.9892 - val_ac
curacy: 0.2101 - val_loss: 2.0810
val. accuracy: 0.218
params: {}
Epoch 1/50
2188/2188  10s 4ms/step - accuracy: 0.1067 - loss: 2.3201 - val_a
ccuracy: 0.0983 - val_loss: 2.2963
Epoch 2/50
2188/2188  9s 4ms/step - accuracy: 0.1330 - loss: 2.2652 - val_ac
curacy: 0.1555 - val_loss: 2.1778
Epoch 3/50
2188/2188  9s 4ms/step - accuracy: 0.1676 - loss: 2.1491 - val_ac
curacy: 0.1796 - val_loss: 2.1270
Epoch 4/50
2188/2188  9s 4ms/step - accuracy: 0.1783 - loss: 2.1103 - val_ac
curacy: 0.2081 - val_loss: 2.0606
Epoch 5/50
2188/2188  9s 4ms/step - accuracy: 0.2312 - loss: 2.0392 - val_ac
curacy: 0.2307 - val_loss: 2.0192
Epoch 6/50
2188/2188  9s 4ms/step - accuracy: 0.2499 - loss: 2.0028 - val_ac
curacy: 0.2417 - val_loss: 2.0202
Epoch 7/50
2188/2188  9s 4ms/step - accuracy: 0.2508 - loss: 1.9857 - val_ac
curacy: 0.2436 - val_loss: 2.0371
val. accuracy: 0.243
```

```

params: {}
Epoch 1/50
2188/2188  10s 5ms/step - accuracy: 0.1386 - loss: 2.2751 - val_a
ccuracy: 0.1687 - val_loss: 2.1562
Epoch 2/50
2188/2188  10s 4ms/step - accuracy: 0.1689 - loss: 2.1456 - val_a
ccuracy: 0.1735 - val_loss: 2.0962
Epoch 3/50
2188/2188  10s 4ms/step - accuracy: 0.1753 - loss: 2.1127 - val_a
ccuracy: 0.1847 - val_loss: 2.0947
Epoch 4/50
2188/2188  10s 4ms/step - accuracy: 0.1803 - loss: 2.1033 - val_a
ccuracy: 0.1852 - val_loss: 2.0587
Epoch 5/50
2188/2188  9s 4ms/step - accuracy: 0.1867 - loss: 2.0771 - val_ac
curacy: 0.1663 - val_loss: 2.1581
Epoch 6/50
2188/2188  9s 4ms/step - accuracy: 0.2015 - loss: 2.0555 - val_ac
curacy: 0.1857 - val_loss: 2.1533
val. accuracy: 0.176
params: {}
Epoch 1/50
2188/2188  10s 4ms/step - accuracy: 0.1563 - loss: 2.2136 - val_a
ccuracy: 0.1932 - val_loss: 2.0672
Epoch 2/50
2188/2188  10s 4ms/step - accuracy: 0.1964 - loss: 2.0646 - val_a
ccuracy: 0.2368 - val_loss: 1.9855
Epoch 3/50
2188/2188  9s 4ms/step - accuracy: 0.2189 - loss: 2.0221 - val_ac
curacy: 0.2271 - val_loss: 2.0373
Epoch 4/50
2188/2188  9s 4ms/step - accuracy: 0.2275 - loss: 1.9961 - val_ac
curacy: 0.2455 - val_loss: 1.9554
Epoch 5/50
2188/2188  9s 4ms/step - accuracy: 0.2289 - loss: 1.9937 - val_ac
curacy: 0.2491 - val_loss: 1.9596
Epoch 6/50
2188/2188  9s 4ms/step - accuracy: 0.2415 - loss: 1.9780 - val_ac
curacy: 0.2289 - val_loss: 1.9910
val. accuracy: 0.239

```

Notice that the number of training epochs is related to the reported validation accuracy. Could it be that our patience value for early stopping is too small?

The message is that you have to be careful about the changing hyperparameter values based on a single test.

A function to perform random search

The problem with grid search is that the number of possible hyperparameter combinations grows exponentially with the number of hyperparameters.


There's lots of active research on better ways to perform hyperparameter tuning. A simple alternative to grid search is random search, in which you just try random hyperparameter settings and hope to get lucky.

A random search example


Look at the hyperparameter settings that were randomly selected.

params: {'act_fun': 'elu', 'layer_sizes': [20]}


Epoch 1/50

2188/2188  **11s** 5ms/step - accuracy: 0.2119 - loss: 2.2281 - val_accuracy: 0.3081 - val_loss: 1.9017

Epoch 2/50

2188/2188  **11s** 5ms/step - accuracy: 0.3074 - loss: 1.9162 - val_accuracy: 0.3349 - val_loss: 1.8683


Epoch 3/50

2188/2188  **11s** 5ms/step - accuracy: 0.3421 - loss: 1.8319 - val_accuracy: 0.3320 - val_loss: 1.9240

Epoch 4/50

2188/2188  **11s** 5ms/step - accuracy: 0.3651 - loss: 1.7804 - val_accuracy: 0.3765 - val_loss: 1.7595


Epoch 5/50

2188/2188  **11s** 5ms/step - accuracy: 0.3798 - loss: 1.7488 - val_accuracy: 0.3813 - val_loss: 1.7550

Epoch 6/50


2188/2188  **11s** 5ms/step - accuracy: 0.3892 - loss: 1.7123 - val_accuracy: 0.3640 - val_loss: 1.7940

Epoch 7/50

2188/2188  **11s** 5ms/step - accuracy: 0.4063 - loss: 1.6845 - val_accuracy: 0.3824 - val_loss: 1.7974

params: {'act_fun': 'elu', 'layer_sizes': [10, 10]}

Epoch 1/50

2188/2188  **15s** 7ms/step - accuracy: 0.2031 - loss: 2.1922 - val_accuracy: 0.2427 - val_loss: 2.2523


Epoch 2/50

2188/2188  **14s** 7ms/step - accuracy: 0.2943 - loss: 1.9310 - val_accuracy: 0.3340 - val_loss: 1.8445


Epoch 3/50

2188/2188  **15s** 7ms/step - accuracy: 0.3204 - loss: 1.8693 - val_accuracy: 0.3280 - val_loss: 1.8378


Epoch 4/50

2188/2188  **14s** 7ms/step - accuracy: 0.3428 - loss: 1.7937 - val_accuracy: 0.3303 - val_loss: 1.8214


Epoch 5/50

2188/2188  **14s** 7ms/step - accuracy: 0.3633 - loss: 1.7724 - val_accuracy: 0.3248 - val_loss: 1.8529


Epoch 6/50

2188/2188  **14s** 7ms/step - accuracy: 0.3727 - loss: 1.7406 - val_accuracy: 0.3557 - val_loss: 1.7738

Epoch 7/50

2188/2188  **14s** 7ms/step - accuracy: 0.3847 - loss: 1.7223 - val_accuracy: 0.3564 - val_loss: 1.7706

Epoch 8/50

2188/2188  **14s** 7ms/step - accuracy: 0.3847 - loss: 1.6993 - val_accuracy: 0.3740 - val_loss: 1.7192

Epoch 9/50

2188/2188  **14s** 7ms/step - accuracy: 0.3916 - loss: 1.7043 - val_accuracy: 0.3573 - val_loss: 1.7774





Epoch 10/50

2188/2188  **14s** 7ms/step - accuracy: 0.3868 - loss: 1.6909 - val_accuracy: 0.3793 - val_loss: 1.7180

Epoch 11/50

2188/2188  **14s** 7ms/step - accuracy: 0.4022 - loss: 1.6579 - val_a

```

ccuracy: 0.3703 - val_loss: 1.7517
Epoch 12/50
2188/2188  14s 7ms/step - accuracy: 0.4021 - loss: 1.6690 - val_a
ccuracy: 0.3773 - val_loss: 1.7240
params: {'act_fun': 'relu', 'layer_sizes': [10, 10]}
Epoch 1/50
2188/2188  10s 4ms/step - accuracy: 0.1033 - loss: 2.3156 - val_a
ccuracy: 0.1045 - val_loss: 2.3027
Epoch 2/50
2188/2188  9s 4ms/step - accuracy: 0.1010 - loss: 2.3027 - val_ac
curacy: 0.1005 - val_loss: 2.3030
Epoch 3/50
2188/2188  9s 4ms/step - accuracy: 0.1012 - loss: 2.3022 - val_ac
curacy: 0.1045 - val_loss: 2.3028

```

Batch normalization

The most commonly-recommended way to apply batch normalization is between the summation part of a neuron and the activation function. See Listings 9.4 and 9.5 in the Chollet text.

Also, in a feedforward network, you would apply batch normalization in one or more hidden layers, not the output layer.

Problem 3. Make modifications to support batch normalization in the hidden layers of your model. The new hyperparameter should be named 'batch_norm' and should take value True or False. Use False as the default.

Test batch normalization

```

params: {'batch_norm': True}
Epoch 1/50

```






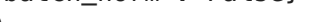

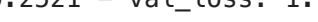

```

/Users/ssogden/miniconda3/envs/cst463/lib/python3.10/site-packages/keras/src/layers/core/input_layer.py:26: UserWarning: Argument `input_shape` is deprecated. Use `shape` instead.
  warnings.warn(

```



```

2188/2188  15s 7ms/step - accuracy: 0.2370 - loss: 2.1053 - val_a
ccuracy: 0.3389 - val_loss: 1.8686
Epoch 2/50
2188/2188  14s 6ms/step - accuracy: 0.3316 - loss: 1.8666 - val_a
ccuracy: 0.2756 - val_loss: 1.9995
Epoch 3/50
2188/2188  14s 6ms/step - accuracy: 0.3474 - loss: 1.8270 - val_a
ccuracy: 0.3779 - val_loss: 1.7381
Epoch 4/50
2188/2188  14s 6ms/step - accuracy: 0.3567 - loss: 1.8007 - val_a
ccuracy: 0.3731 - val_loss: 1.7414
Epoch 5/50
2188/2188  14s 6ms/step - accuracy: 0.3751 - loss: 1.7683 - val_a
ccuracy: 0.3532 - val_loss: 1.8511
params: {'batch_norm': False}
Epoch 1/50
2188/2188  10s 4ms/step - accuracy: 0.1362 - loss: 2.2424 - val_a
ccuracy: 0.1811 - val_loss: 2.1584
Epoch 2/50
2188/2188  10s 4ms/step - accuracy: 0.2171 - loss: 2.0452 - val_a
ccuracy: 0.2521 - val_loss: 1.9971
Epoch 3/50
2188/2188  10s 4ms/step - accuracy: 0.2467 - loss: 2.0008 - val_a
ccuracy: 0.2373 - val_loss: 2.0032
Epoch 4/50
2188/2188  10s 4ms/step - accuracy: 0.2441 - loss: 1.9837 - val_a
ccuracy: 0.2468 - val_loss: 2.0046

```

Problem 4. Try to tune the model to get a high validation accuracy value.

Use grid search and/or random search.

Don't add additional hyperparameters, but feel free to add additional possible values for existing hyperparameters.

Do not modify the code above, but you can redefine existing code as in the last problem.

You will probably want to report on your testing with one or more tables showing hyperparameter values and accuracies. See the `scores_to_dataframe()` example above.

Write your summary here.

Write a summary of your findings for this problem.

Problem 5. Add an additional hyperparameter, and continue tuning.

For example, you could add a new hyperparameter related to optimizer learning rate.

Your goal is to achieve a high validation accuracy value.

Note that a hyperparameter could be used in either `get_model()` or in `evaluate_params()`.

Write your summary here.

Write a summary of your findings for this problem.