

▼ Tuning a Feedforward network (solution)

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.

[Show code](#)

[Show code](#)

▼ 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](#).

Show code

Show code

```
(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.

Show code

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

Some random images

[Show code](#)



Get a subset of the data

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

[Show code](#)

Flatten and scale the data

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

[Show code](#)

Squeeze the dimensions of `y_train`, `y_test`

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

[Show code](#)

[Show code](#)

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

[Show code](#)

[Show code](#)

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_channels` when defining the number of outputs. You'll need to use softmax in the output layer.

[Show code](#)

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.

[Show code](#)

Model: "sequential"

Layer (type)	Output Shape	Param #
dense (Dense)	(None, 10)	30730
dense_1 (Dense)	(None, 10)	110
dense_2 (Dense)	(None, 10)	110

=====
Total params: 30,950
Trainable params: 30,950
Non-trainable params: 0
=====

[Show code](#)

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.

[Show code](#)

```
Epoch 1/10
547/547 [=====] - 5s 5ms/step - loss: 2.3049 - accuracy: 0.1036
Epoch 2/10
547/547 [=====] - 2s 4ms/step - loss: 2.3026 - accuracy: 0.0991
Epoch 3/10
547/547 [=====] - 2s 4ms/step - loss: 2.3026 - accuracy: 0.0997
Epoch 4/10
547/547 [=====] - 2s 4ms/step - loss: 2.3025 - accuracy: 0.1014
Epoch 5/10
547/547 [=====] - 2s 4ms/step - loss: 2.3026 - accuracy: 0.1008
Epoch 6/10
547/547 [=====] - 2s 4ms/step - loss: 2.3026 - accuracy: 0.1018
Epoch 7/10
547/547 [=====] - 2s 4ms/step - loss: 2.3025 - accuracy: 0.0995
Epoch 8/10
547/547 [=====] - 2s 4ms/step - loss: 2.3026 - accuracy: 0.1012
Epoch 9/10
547/547 [=====] - 2s 4ms/step - loss: 2.3026 - accuracy: 0.0994
Epoch 10/10
547/547 [=====] - 2s 4ms/step - loss: 2.3026 - accuracy: 0.1019
```

Use the model to make predictions

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

[Show code](#)

```
array([0.09903407, 0.09910464, 0.09928594, 0.10232144, 0.09699019,
       0.10158176, 0.0968587 , 0.10121251, 0.10280646, 0.10080428],
      dtype=float32)
```

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

[Show code](#)


0.99999994
8

Evaluating the model on new data

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

[Show code](#)

```
313/313 [=====] - 1s 3ms/step - loss: 2.3028 - accuracy: 0.1000  
test accuracy: 0.1000
```



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 patient" and wait a few epochs to see if the value gets lower.

[Show code](#)

[Show code](#)

```
Epoch 1/20  
547/547 [=====] - 3s 4ms/step - loss: 2.3026 - accuracy: 0.1000  
Epoch 2/20  
547/547 [=====] - 2s 4ms/step - loss: 2.3026 - accuracy: 0.0997  
Epoch 3/20  
547/547 [=====] - 2s 4ms/step - loss: 2.3026 - accuracy: 0.1017
```



▼ 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 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".

[Show code](#)

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

[Show code](#)

[Show code](#)

Model: "sequential_1"

Layer (type)	Output Shape	Param #
dense_3 (Dense)	(None, 10)	30730
dense_4 (Dense)	(None, 10)	110
Total params: 30,840		
Trainable params: 30,840		
Non-trainable params: 0		

An example with 10 hidden layers, and ELU activation.

[Show code](#)

[Show code](#)

Model: "sequential_2"

Layer (type)	Output Shape	Param #
dense_5 (Dense)	(None, 10)	30730

dense_6 (Dense)	(None, 5)	55
dense_7 (Dense)	(None, 10)	60

```
=====
Total params: 30,845
Trainable params: 30,845
Non-trainable params: 0
=====
```

[Show code](#)

```
Epoch 1/5
547/547 [=====] - 3s 5ms/step - loss: 2.1920 - accuracy: 0.1807
Epoch 2/5
547/547 [=====] - 2s 4ms/step - loss: 2.0494 - accuracy: 0.2205
Epoch 3/5
547/547 [=====] - 2s 4ms/step - loss: 1.9827 - accuracy: 0.2489
Epoch 4/5
547/547 [=====] - 2s 4ms/step - loss: 1.9358 - accuracy: 0.2654
Epoch 5/5
547/547 [=====] - 2s 4ms/step - loss: 1.9006 - accuracy: 0.2821
```

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

[Show code](#)

```
averaged validation accuracy: 0.266
```

▼ Grid search

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

[Show code](#)

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'.

[Show code](#)

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]]}]
```

[Show code](#)

A grid search example

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

[Show code](#)

```
params: {'act_fun': 'relu', 'layer_sizes': [20]}
Epoch 1/50
2188/2188 [=====] - 9s 4ms/step - loss: 2.3088 - accuracy: 0.00
Epoch 2/50
2188/2188 [=====] - 8s 3ms/step - loss: 2.3028 - accuracy: 0.10
Epoch 3/50
2188/2188 [=====] - 9s 4ms/step - loss: 2.3029 - accuracy: 0.00
params: {'act_fun': 'relu', 'layer_sizes': [10, 10]}
Epoch 1/50
2188/2188 [=====] - 9s 4ms/step - loss: 2.3029 - accuracy: 0.10
Epoch 2/50
2188/2188 [=====] - 11s 5ms/step - loss: 2.3027 - accuracy: 0.00
Epoch 3/50
2188/2188 [=====] - 9s 4ms/step - loss: 2.3027 - accuracy: 0.10
params: {'act_fun': 'elu', 'layer_sizes': [20]}
Epoch 1/50
2188/2188 [=====] - 9s 4ms/step - loss: 2.1320 - accuracy: 0.25
Epoch 2/50
2188/2188 [=====] - 8s 4ms/step - loss: 1.9122 - accuracy: 0.32
Epoch 3/50
2188/2188 [=====] - 8s 4ms/step - loss: 1.8376 - accuracy: 0.35
Epoch 4/50
```


```

2188/2188 [=====] - 8s 4ms/step - loss: 1.7927 - accuracy: 0.36
Epoch 5/50
2188/2188 [=====] - 8s 4ms/step - loss: 1.7549 - accuracy: 0.38
Epoch 6/50
2188/2188 [=====] - 9s 4ms/step - loss: 1.7193 - accuracy: 0.39
params: {'act_fun': 'elu', 'layer_sizes': [10, 10]}
Epoch 1/50
2188/2188 [=====] - 10s 4ms/step - loss: 2.0703 - accuracy: 0.12
Epoch 2/50
2188/2188 [=====] - 8s 4ms/step - loss: 1.9265 - accuracy: 0.28
Epoch 3/50
2188/2188 [=====] - 8s 4ms/step - loss: 1.8761 - accuracy: 0.31
Epoch 4/50
2188/2188 [=====] - 9s 4ms/step - loss: 1.8397 - accuracy: 0.32
Epoch 5/50
2188/2188 [=====] - 9s 4ms/step - loss: 1.8177 - accuracy: 0.33
Epoch 6/50
2188/2188 [=====] - 9s 4ms/step - loss: 1.7941 - accuracy: 0.35
Epoch 7/50
2188/2188 [=====] - 9s 4ms/step - loss: 1.7751 - accuracy: 0.36
Epoch 8/50
2188/2188 [=====] - 8s 4ms/step - loss: 1.7598 - accuracy: 0.36

```

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

[Show code](#)

	act_fun	layer_sizes	optimizer	batch_size	0	
0	relu	[20]	rmsprop	8	0.098733	
1	relu	[10, 10]	rmsprop	8	0.098667	
2	elu	[20]	rmsprop	8	0.354267	
3	elu	[10, 10]	rmsprop	8	0.346067	

▼ 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.

[Show code](#)

```

2188/2188 [=====] - 9s 4ms/step - loss: 2.0619 - accuracy: 0
Epoch 7/50

```

```
2188/2188 [=====] - 9s 4ms/step - loss: 2.0587 - accuracy: 0
Epoch 8/50
2188/2188 [=====] - 8s 4ms/step - loss: 2.0537 - accuracy: 0
Epoch 9/50
2188/2188 [=====] - 9s 4ms/step - loss: 2.0465 - accuracy: 0
Epoch 10/50
2188/2188 [=====] - 9s 4ms/step - loss: 2.0450 - accuracy: 0
Epoch 11/50
2188/2188 [=====] - 9s 4ms/step - loss: 2.0433 - accuracy: 0
val. accuracy: 0.204
params: {}
Epoch 1/50
2188/2188 [=====] - 9s 4ms/step - loss: 2.1451 - accuracy: 0
Epoch 2/50
2188/2188 [=====] - 8s 4ms/step - loss: 2.0773 - accuracy: 0
Epoch 3/50
2188/2188 [=====] - 8s 4ms/step - loss: 2.0654 - accuracy: 0
Epoch 4/50
2188/2188 [=====] - 9s 4ms/step - loss: 2.0575 - accuracy: 0
Epoch 5/50
2188/2188 [=====] - 10s 4ms/step - loss: 2.0498 - accuracy: 0
Epoch 6/50
2188/2188 [=====] - 9s 4ms/step - loss: 2.0460 - accuracy: 0
Epoch 7/50
2188/2188 [=====] - 9s 4ms/step - loss: 2.0429 - accuracy: 0
Epoch 8/50
2188/2188 [=====] - 9s 4ms/step - loss: 2.0400 - accuracy: 0
val. accuracy: 0.192
params: {}
Epoch 1/50
2188/2188 [=====] - 9s 4ms/step - loss: 2.1473 - accuracy: 0
Epoch 2/50
2188/2188 [=====] - 9s 4ms/step - loss: 2.0763 - accuracy: 0
Epoch 3/50
2188/2188 [=====] - 9s 4ms/step - loss: 2.0638 - accuracy: 0
Epoch 4/50
2188/2188 [=====] - 9s 4ms/step - loss: 2.0566 - accuracy: 0
Epoch 5/50
2188/2188 [=====] - 8s 4ms/step - loss: 2.0504 - accuracy: 0
Epoch 6/50
2188/2188 [=====] - 9s 4ms/step - loss: 2.0471 - accuracy: 0
val. accuracy: 0.198
params: {}
Epoch 1/50
2188/2188 [=====] - 10s 4ms/step - loss: 2.1530 - accuracy: 0
Epoch 2/50
2188/2188 [=====] - 9s 4ms/step - loss: 2.0520 - accuracy: 0
Epoch 3/50
2188/2188 [=====] - 9s 4ms/step - loss: 2.0324 - accuracy: 0
Epoch 4/50
2188/2188 [=====] - 9s 4ms/step - loss: 2.0223 - accuracy: 0
Epoch 5/50
2188/2188 [=====] - 8s 4ms/step - loss: 2.0134 - accuracy: 0
Epoch 6/50
2188/2188 [=====] - 9s 4ms/step - loss: 2.0093 - accuracy: 0
Epoch 7/50
```

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.

[Show code](#)

A random search example

Look at the hyperparameter settings that were randomly selected.

[Show code](#)

```
params: {'act_fun': 'elu', 'layer_sizes': [20]}
Epoch 1/50
2188/2188 [=====] - 10s 4ms/step - loss: 2.0432 - accuracy: 0.1
Epoch 2/50
2188/2188 [=====] - 9s 4ms/step - loss: 1.8752 - accuracy: 0.3
Epoch 3/50
2188/2188 [=====] - 9s 4ms/step - loss: 1.8007 - accuracy: 0.3
params: {'act_fun': 'elu', 'layer_sizes': [10, 10]}
Epoch 1/50
2188/2188 [=====] - 10s 4ms/step - loss: 2.0902 - accuracy: 0.1
Epoch 2/50
2188/2188 [=====] - 10s 4ms/step - loss: 1.9304 - accuracy: 0.1
Epoch 3/50
2188/2188 [=====] - 9s 4ms/step - loss: 1.8583 - accuracy: 0.3
Epoch 4/50
2188/2188 [=====] - 8s 4ms/step - loss: 1.8209 - accuracy: 0.3
params: {'act_fun': 'relu', 'layer_sizes': [10, 10]}
Epoch 1/50
2188/2188 [=====] - 8s 4ms/step - loss: 2.3034 - accuracy: 0.1
Epoch 2/50
2188/2188 [=====] - 9s 4ms/step - loss: 2.3028 - accuracy: 0.1
Epoch 3/50
2188/2188 [=====] - 9s 4ms/step - loss: 2.3029 - accuracy: 0.0
```

```
Epoch 4/50
2188/2188 [=====] - 10s 4ms/step - loss: 2.3027 - accuracy: 0.1
Epoch 5/50
2188/2188 [=====] - 10s 5ms/step - loss: 2.3027 - accuracy: 0.1
Epoch 6/50
2188/2188 [=====] - 9s 4ms/step - loss: 2.3027 - accuracy: 0.16
```



▼ 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.

[Show code](#)

[Show code](#)

[Show code](#)

Test batch normalization


[Show code](#)

```
params: {'batch_norm': True}
Epoch 1/50
2188/2188 [=====] - 13s 6ms/step - loss: 1.9791 - accuracy: 0.1
Epoch 2/50
2188/2188 [=====] - 11s 5ms/step - loss: 1.8539 - accuracy: 0.1
Epoch 3/50
2188/2188 [=====] - 11s 5ms/step - loss: 1.8150 - accuracy: 0.1
Epoch 4/50
2188/2188 [=====] - 11s 5ms/step - loss: 1.7766 - accuracy: 0.1
Epoch 5/50
2188/2188 [=====] - 11s 5ms/step - loss: 1.7626 - accuracy: 0.1
params: {'batch_norm': False}
```

```

Epoch 1/50
2188/2188 [=====] - 10s 4ms/step - loss: 2.2105 - accuracy: 0.1
Epoch 2/50
2188/2188 [=====] - 9s 4ms/step - loss: 2.1330 - accuracy: 0.16
Epoch 3/50
2188/2188 [=====] - 9s 4ms/step - loss: 2.1071 - accuracy: 0.17
Epoch 4/50
2188/2188 [=====] - 9s 4ms/step - loss: 2.0972 - accuracy: 0.17
Epoch 5/50
2188/2188 [=====] - 9s 4ms/step - loss: 2.0884 - accuracy: 0.17
Epoch 6/50
2188/2188 [=====] - 9s 4ms/step - loss: 2.0833 - accuracy: 0.17
Epoch 7/50
2188/2188 [=====] - 9s 4ms/step - loss: 2.0800 - accuracy: 0.17
Epoch 8/50
2188/2188 [=====] - 9s 4ms/step - loss: 2.0757 - accuracy: 0.18
Epoch 9/50
2188/2188 [=====] - 8s 4ms/step - loss: 2.0739 - accuracy: 0.18
Epoch 10/50
2188/2188 [=====] - 8s 4ms/step - loss: 2.0723 - accuracy: 0.18
Epoch 11/50
2188/2188 [=====] - 9s 4ms/step - loss: 2.0701 - accuracy: 0.18

```



▼ **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.

[Show code](#)

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()`.

[Show code](#)

Write your summary here.

Write a summary of your findings for this problem.

[Colab paid products](#) - [Cancel contracts here](#)

✓ 0s completed at 12:44 PM

