- Announcements
- Tutorials
- DL
- ML
- CV
- NLP
- 3D
- Get paid to write

More

- Gradient Docs
- ML Showcase
- AI Wiki
- Community Forum
- Paperspace Help Center
- Contact Sales

Sign up Sign in

Search Search in all our content

Press Enter to search

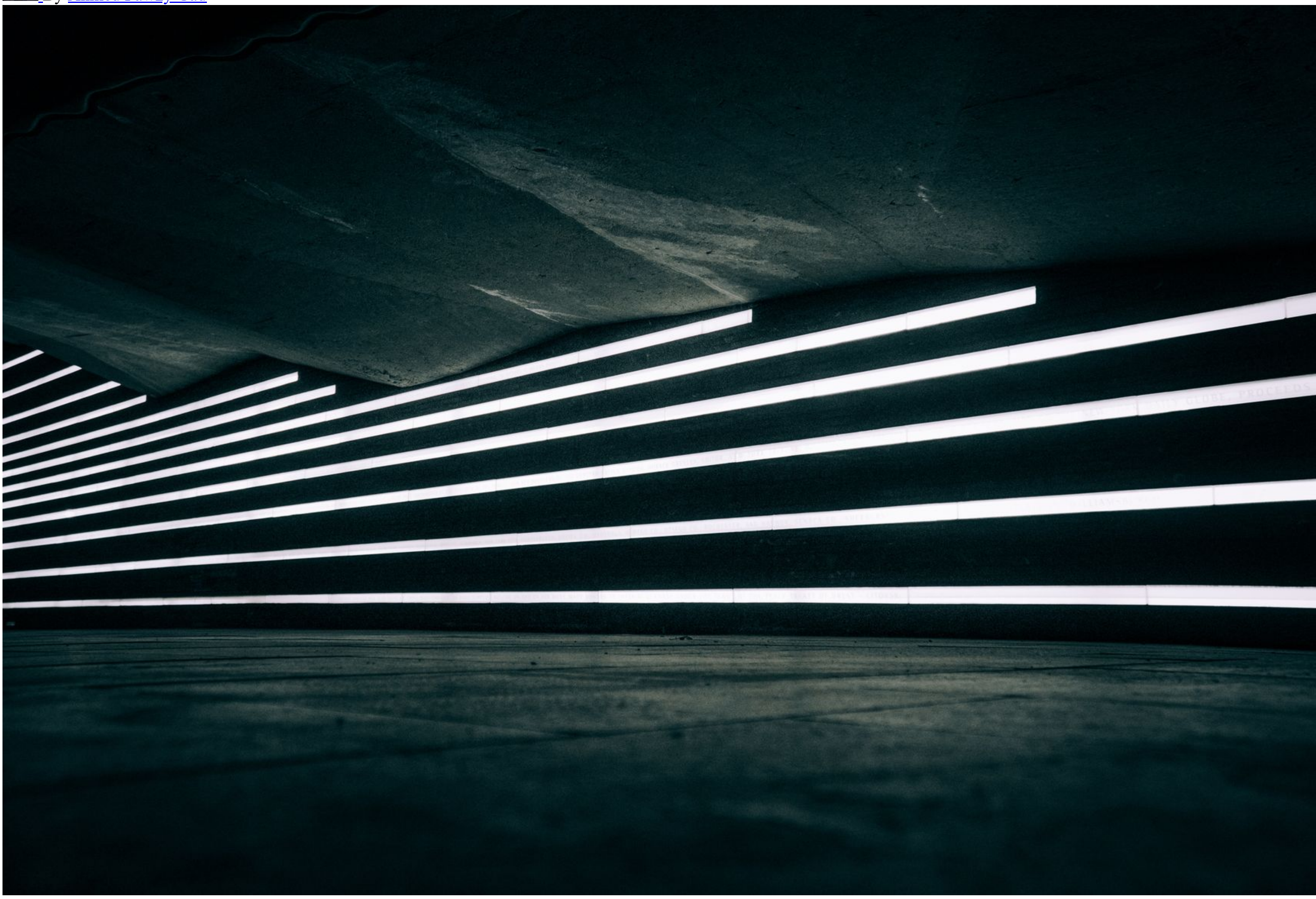
Keras

Working With The Lambda Layer in Keras

In this tutorial we'll cover how to use the Lambda layer in Keras to build, save, and load models which perform custom operations

a year ago • 11 min read

 By Ahmed Fawzy Gad



Keras is a popular and easy-to-use library for building deep learning models. It supports all known type of layers: input, dense, convolutional, transposed convolution, reshape, normalization, dropout, flatten, and activation. Each layer performs a particular operations on the data.

That being said, you might want to perform an operation over the data that is not applied in any of the existing layers, and then these preexisting layer types will not be enough for your task. As a trivial example, imagine you need a layer that performs the operation of adding a fixed number at a given point of the model architecture. Because there is no existing layer that does this, you can build one yourself.

In this tutorial we'll discuss using the `Lambda` layer in Keras. This allows you to specify the operation to be applied as a function. We'll also see how to debug the Keras loading feature when building a model that has lambda layers.

The sections covered in this tutorial are as follows:

- Building a Keras model using the `Functional API`
- Adding a `Lambda` layer
- Passing more than one tensor to the lambda layer
- Saving and loading a model with a lambda layer
- Solving the SystemError while loading a model with a lambda layer

Building a Keras Model Using the `Functional API`

There are three different APIs which can be used to build a model in Keras:

1. Sequential API
2. Functional API
3. Model Subclassing API

You can find more information about each of these in this post, but in this tutorial we'll focus on using the Keras `Functional API` for building a custom model. Since we want to focus on our architecture, we'll just use a simple problem example and build a model which recognizes images in the MNIST dataset.

To build a model in Keras you stack layers on top of one another. These layers are available in the `keras.layers` module (imported below). The module name is prepended by `tensorflow` because we use TensorFlow as a backend for Keras.

```
import tensorflow.keras.layers
```

The first layer to create is the `Input` layer. This is created using the `tensorflow.keras.layers.Input()` class. One of the necessary [illegible] class is the `shape` argument which specifies the shape of each sample in the data that will be used for training. In this tutorial we're just going to use dense layers for starters, and thus the input should be 1-D vector. The `shape` argument is thus assigned a tuple with one value (shown below). The value is 784 because the size of each image in the MNIST dataset is 28 x 28 = 784. An optional `name` argument specifies the name of that layer.

```
input_layer = tensorflow.keras.layers.Input(shape=(784), name="input_layer")
```

The next layer is a dense layer created using the `Dense` class according to the code below. It accepts an argument named `units` to specify the number of neurons in this layer. Note how this layer is connected to the input layer by specifying the name of that layer in parentheses. This is because a layer instance in the functional API is callable on a tensor, and also returns a tensor.

```
dense_layer_1 = tensorflow.keras.layers.Dense(units=500, name="dense_layer_1")(input_layer)
```

Following the dense layer, an activation layer is created using the `ReLU` class according to the next line.

```
activ_layer_1 = tensorflow.keras.layers.ReLU(name="activ_layer_1")(dense_layer_1)
```

Another couple of dense-ReLu layers are added according to the following lines.

```
dense_layer_2 = tensorflow.keras.layers.Dense(units=250, name="dense_layer_2")(activ_layer_1)
activ_layer_2 = tensorflow.keras.layers.ReLU(name="relu_layer_2")(dense_layer_2)

dense_layer_3 = tensorflow.keras.layers.Dense(units=20, name="dense_layer_3")(activ_layer_2)
activ_layer_3 = tensorflow.keras.layers.ReLU(name="relu_layer_3")(dense_layer_3)
```

The next line adds the last layer to the network architecture according to the number of classes in the MNIST dataset. Because the MNIST dataset includes 10 classes (one for each number), the number of units used in this layer is 10.

```
dense_layer_4 = tensorflow.keras.layers.Dense(units=10, name="dense_layer_4")(activ_layer_3)
```

To return the score for each class, a `softmax` layer is added after the previous dense layer according to the next line.

```
output_layer = tensorflow.keras.layers.Softmax(name="output_layer")(dense_layer_4)
```

We've now connected the layers but the model is not yet created. To build a model we must now use the `Model` class, as shown below. The first two arguments it accepts represent the input and output layers.

```
model = tensorflow.keras.models.Model(input_layer, output_layer, name="model")
```

Before loading the dataset and training the model, we have to compile the model using the `compile()` method.

```
model.compile(optimizer=tensorflow.keras.optimizers.Adam(lr=0.0005), loss="categorical_crossentropy")
```

Using `model.summary()` we can see an overview of the model architecture. The input layer accepts a tensor of shape (None, 784) which means that each sample must be reshaped into a vector of 784 elements. The output `Softmax` layer returns 10 numbers, each being the score for that class of the MNIST dataset.

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_layer (InputLayer)     [(None, 784)]             0
_________________________________________________________________
dense_layer_1 (Dense)        (None, 500)               392500
_________________________________________________________________
relu_layer_1 (ReLU)          (None, 500)               0
_________________________________________________________________
dense_layer_2 (Dense)        (None, 250)               125250
_________________________________________________________________
relu_layer_2 (ReLU)          (None, 250)               0
_________________________________________________________________
dense_layer_3 (Dense)        (None, 20)                12550
_________________________________________________________________
relu_layer_3 (ReLU)          (None, 20)                0
_________________________________________________________________
dense_layer_4 (Dense)        (None, 10)                510
_________________________________________________________________
output_layer (Softmax)       (None, 10)                0
=================================================================
Total params: 530,810
Trainable params: 530,810
Non-trainable params: 0
_________________________________________________________________
```

Now that we've built and compiled the model, let's see how the dataset is prepared. First we'll load MNIST from the `keras.datasets` module, got their data type changed to `float64` because this makes training the network easier than leaving its values in the 0-255 range, and finally reshaped so that each sample is a vector of 784 elements.

```
(x_train, y_train), (x_test, y_test) = tensorflow.keras.datasets.mnist.load_data()

x_train = x_train.astype(numpy.float64) / 255.0
x_test = x_test.astype(numpy.float64) / 255.0

x_train = x_train.reshape((x_train.shape[0], numpy.prod(x_train.shape[1:])))
x_test = x_test.reshape((x_test.shape[0], numpy.prod(x_test.shape[1:])))
```

Because the used loss function in the `compile()` method is `categorical_crossentropy`, the labels of the samples should be on hot encoded according to the next code.

```
y_test = tensorflow.keras.utils.to_categorical(y_test)
y_train = tensorflow.keras.utils.to_categorical(y_train)
```

Finally, the model training starts using the `fit()` method.

```
model.fit(x_train, y_train, epochs=20, batch_size=256, validation_data=(x_test, y_test))
```

At this point, we have created the model architecture using the already existing types of layers. The next section discusses usi‿ tl ... ıer

Using Lambda Layers

Let's say that after the dense layer named `dense_layer_3` we'd like to do some sort of operation on the tensor, such as adding the value 2 to each element. How can we do that? None of the existing layers does this, so we'll have to build a new layer ourselves. Fortunately, the `Lambda` layer exists for precisely that purpose. Let's discuss how to use it.

Start by building the function that will do the operation you want. In this case, a function named `custom_layer` is created as follows. It just accepts the input tensor(s) and returns another tensor as output. If more than one tensor is to be passed to the function, then they will be passed as a list.

In this example just a single tensor is fed as input, and 2 is added to each element in the input tensor.

```
def custom_layer(tensor):
    return tensor + 2
```

After building the function that defines the operation, next we need to create the lambda layer using the `Lambda` class as defined in the next line. In this case, only one tensor is fed to the `custom_layer` function because the lambda layer is callable on the single tensor returned by the dense layer named `dense_layer_3`.

```
lambda_layer = tensorflow.keras.layers.Lambda(custom_layer, name="lambda_layer")(dense_layer_3)
```

Here is the code that builds the full network after using the lambda layer.

```
input_layer = tensorflow.keras.layers.Input(shape=(784), name="input_layer")

dense_layer_1 = tensorflow.keras.layers.Dense(units=500, name="dense_layer_1")(input_layer)
activ_layer_1 = tensorflow.keras.layers.ReLU(name="relu_layer_1")(dense_layer_1)

dense_layer_2 = tensorflow.keras.layers.Dense(units=250, name="dense_layer_2")(activ_layer_1)
activ_layer_2 = tensorflow.keras.layers.ReLU(name="relu_layer_2")(dense_layer_2)

dense_layer_3 = tensorflow.keras.layers.Dense(units=20, name="dense_layer_3")(activ_layer_2)

def custom_layer(tensor):
    return tensor + 2

lambda_layer = tensorflow.keras.layers.Lambda(custom_layer, name="lambda_layer")(dense_layer_3)

activ_layer_3 = tensorflow.keras.layers.ReLU(name="relu_layer_3")(lambda_layer)

dense_layer_4 = tensorflow.keras.layers.Dense(units=10, name="dense_layer_4")(activ_layer_3)
output_layer = tensorflow.keras.layers.Softmax(name="output_layer")(dense_layer_4)

model = tensorflow.keras.models.Model(input_layer, output_layer, name="model")
```

In order to see the tensor before and after being fed to the lambda layer we'll create two new models in addition to the previous one. We'll call these `before_lambda_model` and `after_lambda_model`. Both models use the input layer as their inputs, but the output layer differs. The `before_lambda_model` model returns the output of `dense_layer_3` which is the layer that exists exactly before the lambda layer. The output of the `after_lambda_model` model is the output from the lambda layer named `lambda_layer`. By doing this, we can see the input before and the output after applying the lambda layer.

```
before_lambda_model = tensorflow.keras.models.Model(input_layer, dense_layer_3, name="before_lambda_model")
after_lambda_model = tensorflow.keras.models.Model(input_layer, lambda_layer, name="after_lambda_model")
```

The complete code that builds and trains the entire network is listed below.

```
import tensorflow.keras.layers
import tensorflow.keras.models
import tensorflow.keras.optimizers
import tensorflow.keras.datasets
import tensorflow.keras.utils
import tensorflow.keras.backend
import numpy

input_layer = tensorflow.keras.layers.Input(shape=(784), name="input_layer")

dense_layer_1 = tensorflow.keras.layers.Dense(units=500, name="dense_layer_1")(input_layer)
activ_layer_1 = tensorflow.keras.layers.ReLU(name="relu_layer_1")(dense_layer_1)

dense_layer_2 = tensorflow.keras.layers.Dense(units=250, name="dense_layer_2")(activ_layer_1)
activ_layer_2 = tensorflow.keras.layers.ReLU(name="relu_layer_2")(dense_layer_2)

dense_layer_3 = tensorflow.keras.layers.Dense(units=20, name="dense_layer_3")(activ_layer_2)

before_lambda_model = tensorflow.keras.models.Model(input_layer, dense_layer_3, name="before_lambda_model")

def custom_layer(tensor):
    return tensor + 2

lambda_layer = tensorflow.keras.layers.Lambda(custom_layer, name="lambda_layer")(dense_layer_3)
after_lambda_model = tensorflow.keras.models.Model(input_layer, lambda_layer, name="after_lambda_model")

activ_layer_3 = tensorflow.keras.layers.ReLU(name="relu_layer_3")(lambda_layer)

dense_layer_4 = tensorflow.keras.layers.Dense(units=10, name="dense_layer_4")(activ_layer_3)
output_layer = tensorflow.keras.layers.Softmax(name="output_layer")(dense_layer_4)

model = tensorflow.keras.models.Model(input_layer, output_layer, name="model")

model.compile(optimizer=tensorflow.keras.optimizers.Adam(lr=0.0005), loss="categorical_crossentropy")
model.summary()

(x_train, y_train), (x_test, y_test) = tensorflow.keras.datasets.mnist.load_data()

x_train = x_train.astype(numpy.float64) / 255.0
x_test = x_test.astype(numpy.float64) / 255.0

x_train = x_train.reshape((x_train.shape[0], numpy.prod(x_train.shape[1:])))
x_test = x_test.reshape((x_test.shape[0], numpy.prod(x_test.shape[1:])))

y_test = tensorflow.keras.utils.to_categorical(y_test)
y_train = tensorflow.keras.utils.to_categorical(y_train)

model.fit(x_train, y_train, epochs=20, batch_size=256, validation_data=(x_test, y_test))
```

Note that you do not have to compile or train the 2 newly created models because their layers are actually reused from the main m Af model is trained, we can use the `predict()` method for returning the outputs of the `before_lambda_model` and `after_lambda_model`

```
p = model.predict(x_train)

m1 = before_lambda_model.predict(x_train)
m2 = after_lambda_model.predict(x_train)
```

The next code just prints the outputs of the first 2 samples. As you can see, each element returned from the `m2` array is actually the result of `m1` after adding 2. This is exactly the operation we applied in our custom lambda layer.

```
print(m1[0, :])
print(m2[0, :])
```

```
[ 14.420735    8.872794   25.369402    1.4622561   5.672293    2.5202641
 -14.753801   -3.8822086  -1.0581762  -6.4336205  13.342142   -3.0627508
  -5.694006   -6.557313   -1.6567478  -3.8457105  11.891999   20.581928
   2.669979   -8.092522 ]
[ 16.420734   10.872794   27.369402    3.462256    7.672293
   4.520264  -12.753801   -1.8822086   0.94182384  -4.4336205
  15.342142   -1.0627508   -3.694006   -4.557313    0.34325218
  -1.8457105  13.891999   22.581928    4.669979   -6.0925217 ]
```

In this section the lambda layer was used to do an operation over a single input tensor. In the next section we see how we can pass two input tensors to this layer.

Passing More Than One Tensor to the Lambda Layer

Assume that we want to do an operation that depends on the two layers named `dense_layer_3` and `relu_layer_3`. In this case we have to call the lambda layer while passing two tensors. This is simply done by creating a list with all of these tensors, as given in the next line.

```
lambda_layer = tensorflow.keras.layers.Lambda(custom_layer, name="lambda_layer")([dense_layer_3, activ_layer_3])
```

This list is passed to the `custom_layer()` function and we can fetch the individual layers simply according to the next code. It just adds these two layers together. There is actually layer in Keras named `Add` that can be used for adding two layers or more, but we are just presenting how you could do it yourself in case there's another operation not supported by Keras.

```
def custom_layer(tensor):
    tensor1 = tensor[0]
    tensor2 = tensor[1]
    return tensor1 + tensor2
```

The next code builds three models: two for capturing the outputs from the `dense_layer_3` and `activ_layer_3` passed to the lambda layer, and another one for capturing the output from the lambda layer itself.

```
before_lambda_model1 = tensorflow.keras.models.Model(input_layer, dense_layer_3, name="before_lambda_model1")
before_lambda_model2 = tensorflow.keras.models.Model(input_layer, activ_layer_3, name="before_lambda_model2")

lambda_layer = tensorflow.keras.layers.Lambda(custom_layer, name="lambda_layer")([dense_layer_3, activ_layer_3])
after_lambda_model = tensorflow.keras.models.Model(input_layer, lambda_layer, name="after_lambda_model")
```

To see the outputs from the `dense_layer_3`, `activ_layer_3`, and `lambda_layer` layers, the next code predicts their outputs and prints it.

```
m1 = before_lambda_model1.predict(x_train)
m2 = before_lambda_model2.predict(x_train)
m3 = after_lambda_model.predict(x_train)

print(m1[0, :])
print(m2[0, :])
print(m3[0, :])
```

```
[ 1.773366   -3.4378722   0.22042789 11.220362    3.4020965  14.487111
  4.239182   -6.8589864  -6.428128   -5.477719   -8.799093    7.264849
 17.503246   -6.809489   -6.846208   16.094025   24.483786   -7.084775
 17.341183   20.311539  ]
[ 1.773366    0.          0.22042789 11.220362    3.4020965  14.487111
  4.239182    0.          0.          0.          0.          7.264849
 17.503246    0.          0.         16.094025   24.483786    0.
 17.341183   20.311539  ]
[ 3.546732   -3.4378722   0.44085577 22.440723    6.804193   28.974222
  8.478364   -6.8589864  -6.428128   -5.477719   -8.799093   14.529698
 35.006493   -6.809489   -6.846208   32.18805    48.96757    -7.084775
 34.682365   40.623077  ]
```

Using the lambda layer is now clear. The next section discusses how you can save and load a model that uses a lambda layer.

Saving and Loading a Model With a Lambda Layer

In order to save a model (whether it uses a lambda layer or not) the `save()` method is used. Assuming we are just interested in saving the main model, here's the line that saves it.

```
model.save("model.h5")
```

We can also load the saved model using the `load_model()` method, as in the next line.

```
loaded_model = tensorflow.keras.models.load_model("model.h5")
```

Hopefully, the model could be successfully loaded. Unfortunately there are some issues in Keras that may result in the `SystemError: unknown opcode` while loading a model with a lambda layer. It might be due to building the model using a Python version and using it in another version. We are going to discuss the solution in the next section.

Solving The SystemError While Loading a Model with a Lambda Layer

To solve this issue we're not going to save the model in the way discussed above. Instead, we'll save the model weights using the `save_weights()` method.

Now we've only saved the weights. What about the model architecture? The model architecture will be recreated using the code. Why not save the model architecture as a JSON file and then load it again? The reason is that the error persists after loading the architecture.

In summary, the trained model weights will be saved, the model architecture will be reproduced using the code, and finally the we

The weights of the model can be saved using the next line.

```
model.save_weights('model_weights.h5')
```

Here's the code that reproduces the model architecture. The `model` will not be trained, but the saved weights will be assigned to it again.

```
input_layer = tensorflow.keras.layers.Input(shape=(784), name="input_layer")

dense_layer_1 = tensorflow.keras.layers.Dense(units=500, name="dense_layer_1")(input_layer)
activ_layer_1 = tensorflow.keras.layers.ReLU(name="relu_layer_1")(dense_layer_1)

dense_layer_2 = tensorflow.keras.layers.Dense(units=250, name="dense_layer_2")(activ_layer_1)
activ_layer_2 = tensorflow.keras.layers.ReLU(name="relu_layer_2")(dense_layer_2)

dense_layer_3 = tensorflow.keras.layers.Dense(units=20, name="dense_layer_3")(activ_layer_2)
activ_layer_3 = tensorflow.keras.layers.ReLU(name="relu_layer_3")(dense_layer_3)

def custom_layer(tensor):
    tensor1 = tensor[0]
    tensor2 = tensor[1]

    epsilon = tensorflow.keras.backend.random_normal(shape=tensorflow.keras.backend.shape(tensor1), mean=0.0, stddev=1.0)
    random_sample = tensor1 + tensorflow.keras.backend.exp(tensor2/2) * epsilon
    return random_sample

lambda_layer = tensorflow.keras.layers.Lambda(custom_layer, name="lambda_layer")([dense_layer_3, activ_layer_3])

dense_layer_4 = tensorflow.keras.layers.Dense(units=10, name="dense_layer_4")(lambda_layer)
after_lambda_model = tensorflow.keras.models.Model(input_layer, dense_layer_4, name="after_lambda_model")

output_layer = tensorflow.keras.layers.Softmax(name="output_layer")(dense_layer_4)

model = tensorflow.keras.models.Model(input_layer, output_layer, name="model")

model.compile(optimizer=tensorflow.keras.optimizers.Adam(lr=0.0005), loss="categorical_crossentropy")
```

Here's how the saved weights are loaded using the `load_weights()` method, and assigned to the reproduced architecture.

```
model.load_weights('model_weights.h5')
```

Conclusion

This tutorial discussed using the `Lambda` layer to create custom layers which do operations not supported by the predefined layers in Keras. The constructor of the `Lambda` class accepts a function that specifies how the layer works, and the function accepts the tensor(s) that the layer is called on. Inside the function, you can perform whatever operations you want and then return the modified tensors.

Although Keras has an issue with loading models that use the lambda layer, we also saw how to solve this simply by saving the trained model weights, reproducing the model architecture using code, and loading the weights into this architecture.

3 replies

masoudbn May '20

Thanks Ahmed!
It's a great post and very useful.
It really helped me.

ahmedgad Jul '20

Hi,

Glad the post is useful and left a good impression 🙂

Regards,
Ahmed

asquare1312 15h

Mam, Can you please explain how backpropogation is being done in Lambda Layer

- Tags:
- Keras
- Neural Network
- Deep Learning

Spread the word

- Share
- Tweet
- Share
- Copy
- Email

https://blog.paperspace.com/

public

Next article

The Future of ML: Unsupervised Learning, Reinforcement Learning, or Something Else?

public

Previous article

Understanding GauGAN Part 4: Debugging Training & Deciding If GauGAN Is Right For You

Keep reading

public

Object Detection Using Directed Mask R-CNN With Keras

25 days ago • 21 min read

public

Attention Mechanisms in Recurrent Neural Networks (RNNs) With Keras

a month ago • 22 min read

public

[Federated Learning With Keras]

2 months ago • 12 min read

Subscribe to our newsletter

Stay updated with Paperspace Blog by signing up for our newsletter.

Your email address Your email address Join now

🎉 Awesome! Now check your inbox and click the link to confirm your subscription.

Please enter a valid email address

Oops! There was an error sending the email, please try later

PaperspaceBlog

Main links

- Announcements
- Tutorials
- DL
- ML
- CV
- NLP
- 3D
- 💎 Get paid to write

Secondary links

- Gradient Docs
- ML Showcase
- AI Wiki
- Community Forum
- Paperspace Help Center
- Contact Sales

Social links

- Facebook
- Twitter