# APPENDIX A

# Intro to Keras

In this appendix, you will be introduced to the Keras framework along with the functionality that it offers. You will also take a look at using the back end, which is TensorFlow in this case, to perform low-level operations all using Keras.

Regarding the setup, we use

- tensorflow-gpu version 1.10.0
- keras version 2.0.8
- torch version 0.4.1 (this is PyTorch)
- CUDA version 9.0.176
- cuDNN version 7.3.0.29

## What Is Keras?

Keras is a high-level, deep learning library for Python, running with TensorFlow, CNTK, or Theanos as the **back end**. The back end can basically be thought of as the "engine" that does all of the work, and Keras is the rest of the car, including the software that interfaces with the engine.

In other words, Keras being high-level means that it abstracts away much of the intricacies of a framework like TensorFlow. You only need to write a few lines of code to have a deep learning model ready to train and ready to use. In contrast, TensorFlow being more of a low-level framework means you have much more added syntax and functionality to define the extra work that Keras abstracts away for you. At the same time, TensorFlow and PyTorch also allow for much more flexibility if you know what you're doing.




S. Alla and S. K. Adari, *Beginning Anomaly Detection Using Python-Based Deep Learning*,
https://doi.org/10.1007/978-1-4842-5177-5

TensorFlow and PyTorch allow you to manipulate individual **tensors** (similar to matrices, but they aren't limited to two dimensions; they can range from vectors to matrices to n-dimensional objects) to create custom neural network layers, and to create new neural network architectures that include custom layers.

With that being said, Keras allows you to do the same things as TensorFlow and PyTorch do, but you will have to import the back end itself (which in this case is TensorFlow) to perform any of the low-level operations. This is basically the same thing as working with TensorFlow itself since you're using the TensorFlow syntax through Keras, so you still need to be knowledgeable about TensorFlow syntax and functionality.

In the end, if you're not doing research work that requires you to create a new type of model, or to manipulate the tensors directly, simply use Keras. It's a much easier framework to use for beginners, and it will go a long way until you become sufficiently advanced enough that you need the low-level functionality that TensorFlow or PyTorch offers. And even then, you can still use TensorFlow (or whatever back end you're using) through Keras if you need to do any low-level work. One thing to note is that Keras has actually been integrated into TensorFlow, so you can access Keras through TensorFlow itself, but for the purpose of this appendix, we will use the Keras API to showcase the Keras functionality, and the TensorFlow back end through Keras to demonstrate the low-level operations that are analogous to PyTorch.

# Using Keras

When using Keras, you will most likely import the necessary packages, load the data, process it, and then pass it into the model. In this section, we will cover model creation in Keras, the different layers available, several submodules of Keras, and how to use the back end to perform tensor operations.

If you'd like to learn Keras even more in depth, feel free to check out the official documentation. We only cover the basic essentials that you need to know about Keras, so if you have further questions or would like to learn more, we recommend you to explore the documentation.

For details on implementation, Keras is available on GitHub at `https://github.com/keras-team/keras/tree/c2e36f369b411ad1d0a40ac096fe35f73b9dffd3`.

The official documentation is available at `https://keras.io/`.

## Model Creation

In Keras, you can build a **sequential model**, or a **functional model**.

The **sequential model** is built as shown in Figure A-1.

```
In [2]:
1  ### Sequential model
2
3  import keras
4  from keras.models import Sequential
5  from keras.layers import Dense
6
7  seq_model = Sequential()
8  seq_model.add(Dense(16, input_shape=(8,)))
9  seq_model.add(Dense(32, activation='relu'))
10 seq_model.add(Dense(16, activation='softmax'))
11
```

***Figure A-1.*** *Code defining a sequential model in Keras*

Once you've defined a sequential model, you can simply add layers to it by calling `model_name.add()`, where the layer itself is the parameter. Once you've finished adding all of the layers that you want, you are ready to compile and train the model on whatever data you have.

Now, let's look at the **functional model,** the format of which is what you've used in the book thus far (see Figure A-2).

```
In [5]:
1  ### Functional model
2
3  import keras
4  from keras.models import Model
5  from keras.layers import Input, Dense
6
7  input_layer = Input(shape=(8,))
8  dense_1 = Dense(32, activation='relu')(input_layer)
9  output_layer = Dense(16, activation='softmax')(dense_1)
10
11 func_model = Model(input_layer, output_layer)
```

***Figure A-2.*** *Code defining a functional model in Keras*

The functional model allows you to have more flexibility in how you define your neural network. With it, you can connect layers to any other layer that you want, instead of being limited to just the previous layer like in the sequential model. This allows you to share a layer with multiple other layers or even reuse the same layer, allowing you to create more complicated models.

Once you're done defining all of you layers, you simply need to call `Model()` with your input and output parameters respectively to finish your whole model. Now, you can continue onwards to compiling and training your model.

## Model Compilation and Training

In most cases, the code to compile your model will look something like Figure A-3.

```
In [ ]:  1 model.compile(optimizer="",
         2                       loss="",
         3                       metrics="")
```

***Figure A-3.*** *Code to compile a model in Keras*

However, there are many more parameters to consider:

- **optimizer**: Passes in the name of the optimizer in the string or an instance of the optimizer (you call the optimizer with whatever parameters you like. We will elaborate on this further below in the **Optimizers** section.)
- **loss**: Passes in the name of the loss function or the function itself. We elaborate on what we mean by this below in the **Losses** section.
- **metrics**: Passes in the list of metrics that you want the model to evaluate during the training and testing processes. Check out the **Metrics** section for more details on what metrics you can use.
- **loss_weights**: If you have multiple outputs and multiple losses, the model evaluates based on the total loss. The loss_weights are a list or dictionary that determines how much each loss factors into the overall, combined loss. With the new weights, the overall loss is now the weighted sum of all losses.
- **sample_weight_mode**: If your data has 2D weights with timestep-wise sample weighting, then you should pass in `"temporal"`. Otherwise, `None` defaults to 1D sample-wise weights. You can also pass a list or dictionary of sample_weight_modes if your model has multiple outputs. One thing to note is that you need at least a 3D output, with one dimension being time.

- **weighted_metrics**: A list of metrics for the model to evaluate and weight using sample_weight or class_weight during the training and testing processes.

After compiling the model, you can also call a function to **save** your model as in Figure A-4.

```
In [ ]:
from keras.callbacks import ModelCheckpoint

checkpointer = ModelCheckpoint(filepath="saved_model.h5",
                               verbose=0,
                               save_best_only=True)
```

***Figure A-4.*** *A callback to save the model to some file path*

Here are the set of parameters associated with `ModelCheckpoint()`:

- **filepath**: The path where you want to save the model file. Typing just "saved_model.h5" saves it in the same directory.
- **monitor**: The quantity that you want the model to monitor. By default, it's set to "val_loss".
- **verbose**: Sets verbosity to 0 or 1. It's set to 0 by default.
- **save_best_only**: If set to True, then the model with the best performance according to the quantity monitored will be saved.
- **save_weights_only**: If set to True, then only the weights will be saved. Essentially, if True, model.save_weights(filepath), else model.save(filepath).
- **mode**: Can choose between auto, min, or max. If `save_best_only` is True, then you should pick a choice that would suit the monitored quantity best. If you chose val_acc for monitor, then you want to pick max for mode, and if you choose val_loss for monitor, pick min for mode.
- **period**: How many epochs there are between each checkpoint.

Now, you can train your model using code similar to Figure A-5.

```
In [ ]:
model.fit(x, y,
          batch_size=128,
          epochs=25,
          verbose=1,
          validation_data=(x_t, y_t),
          callbacks = [checkpointer])
```

***Figure A-5.*** *Code to train the model*

The `model.fit()` function has a big list of parameters:

- **x**: This is a Numpy array representing the training data. If you have multiple inputs, then this is a list of Numpy arrays that are all training data.
- **y**: This is a Numpy array that represents the target or label data. Again, if you have multiple outputs, then this is a list of target data Numpy arrays.
- **batch_size**: Set to 32 by default. This is the integer number of samples to run through the network before updating the gradients.
- **epochs**: An integer value dictating how many iterations for the entire x and y data to pass through the network.
- **verbose**: 0 makes it train without outputting anything, 1 shows a progress bar and the metrics, and 2 shows one line per epoch. Check the figures below for exactly what each value does:

Verbosity 1 (Figure A-6)

```
In [16]:
TCN.fit(x_train, y_train,
          batch_size=128,
          epochs=25,
          verbose=1,
          validation_data=(x_test, y_test),
          callbacks = [checkpointer])

Train on 6295 samples, validate on 4197 samples
Epoch 1/25
6295/6295 [==============================] - 0s - loss: 0.0620 - acc: 0.9911 - val_loss: 0.0641 - val_acc: 0.9900
Epoch 2/25
6295/6295 [==============================] - 0s - loss: 0.0656 - acc: 0.9895 - val_loss: 0.0655 - val_acc: 0.9890
Epoch 3/25
6295/6295 [==============================] - 0s - loss: 0.0622 - acc: 0.9905 - val_loss: 0.0630 - val_acc: 0.9907
Epoch 4/25
3712/6295 [================>.............] - ETA: 0s - loss: 0.0637 - acc: 0.9903
```

***Figure A-6.*** *The training function with verbosity 1*

Verbosity 2 (Figure A-7)

```
In [17]: TCN.fit(x_train, y_train,
                 batch_size=128,
                 epochs=25,
                 verbose=2,
                 validation_data=(x_test, y_test),
                 callbacks = [checkpointer])

Train on 6295 samples, validate on 4197 samples
Epoch 1/25
1s - loss: 0.0633 - acc: 0.9900 - val_loss: 0.0639 - val_acc: 0.9914
Epoch 2/25
0s - loss: 0.0621 - acc: 0.9905 - val_loss: 0.0626 - val_acc: 0.9914
Epoch 3/25
0s - loss: 0.0639 - acc: 0.9897 - val_loss: 0.0637 - val_acc: 0.9912
Epoch 4/25
```

***Figure A-7.*** *The training function with verbosity 2*

- **callbacks**: A list of `keras.callbacks.Callback` instances. Remember the ModelCheckpoints instance defined earlier as "checkpointer"? This is where you include it. To see how it's done, refer to one of the above figures that showcase the `model.fit()` function being called.
- **validation_split**: A float value between 0 and 1 that tells the model how much of the training data should be used as validation data.
- **validation_data**: A tuple (x_val, y_val) or (x_val, y_val, val_sample_weights) with variable parameters that pass the validation data to the model, and optionally, the val_sample_weights as well. This also overrides validation_split, so use one or the other.
- **shuffle**: A Boolean that tells the model whether or not to shuffle the training data before each epoch, or pass in a string for "batch", meaning it shuffles in batch-sized chunks.
- **class_weight**: (optional) A dictionary that tells the model how to weigh certain classes in the training process. You can use it to weigh under-represented classes higher, for example.
- **sample_weight**: (optional) A Numpy array of weights that have a 1:1 map between the training samples and the weight array you passed in. If you have temporal data (an extra time dimension), pass in a 2D

array with a shape (samples, sequence_length) to apply these weights to each timestep of the samples. Don't forget to set "temporal" for sample_weight_mode in `model.compile()`.

- **initial_epoch**: An integer that tells the model what epoch to start training at (can be used when resuming training).
- **steps_per_epoch**: The number of steps, or batches of samples, for the model to take before completing one epoch.
- **validation_steps**: (Only if you specify steps_per_epoch.) The number of steps to take (number of batches of samples) to use for validation before stopping.
- **validation_freq**: (Only if you pass in validation data.) If you pass in **n**, it runs validation every **n** epochs. If you pass in [**a, e, h**], it runs validation after epoch **a**, epoch **e**, and epoch **h**.

## Model Evaluation and Prediction

After training the model, you can not only evaluate its performance on some test data, but you can make predictions and use the output for any other application you want. Previously, you've used the predictions to generate AUC scores to help better evaluate the model (accuracy is not the best metric to judge model performance by), but you can use these predictions in any way you want, especially if the model's really good at its job.

The code to evaluate your model on some test data might look similar to Figure A-8.

```
In [ ]:  1  model.evaluate(x, y, verbose=0)
```

***Figure A-8.*** *Code to evaluate the model given x and y data sets*

For model.evaluate(), the parameters are

- **x**: The Numpy array representing the test data. Pass in a list of Numpy arrays if the model has multiple inputs.
- **y**: The Numpy array of target or label data that is a part of the test data. If there are multiple inputs, pass in a list of Numpy arrays.

- **batch_size**: If none is specified, the default is 32. This parameter expects an integer value that dictates how many samples there are per evaluation step.
- **verbose**: If set to 0, no output is shown. If set to 1, the progress bar is shown and looks like Figure A-9.

```
In [19]:
score = TCN.evaluate(x_test, y_test, verbose=1)
print('Test loss:', score[0])
print('Test accuracy:', score[1])

4197/4197 [==============================] - 1s
Test loss: 0.06095811567112381
Test accuracy: 0.9914224446032881
```

***Figure A-9.*** *The evaluate function with verbosity 1*

- **sample_weight**: (optional) A Numpy array of weights for each of the test samples. Again, either a 1:1 map between the sample and the weights, unless it's temporal data. If you have temporal data (an extra time dimension), pass in a 2D array with a shape (samples, sequence_length) to apply these weights to each timestep of the samples. Don't forget to set "temporal" for sample_weight_mode in `model.compile()`.
- **steps**: If None, then ignored. Otherwise, it's the integer parameter **n** number of steps (batches of samples) before declaring the evaluation as done.
- **callbacks**: Works the same way as the callbacks parameter for `model.fit()`.

Finally, to make predictions, you can run code similar to Figure A-10.

```
In [ ]:
model.predict(x)
```

***Figure A-10.*** *The prediction function generates predictions given some data set x*

In this case, the parameters are

- **x**: The Numpy array representing the prediction data. Pass in a list of Numpy arrays if the model has multiple inputs.
- **batch_size**: If none is specified, the default is 32. This parameter expects an integer value that dictates how many samples there are per batch.
- **verbose**: Either a 0 or 1.
- **steps**: How many steps to take (batches of samples) before finishing the prediction process. This is ignored if None is passed in.
- **callbacks**: Works the same way as the callbacks parameter for `model.fit()`.

One more thing to mention: If you've saved a model, you can load it again by calling the code in Figure A-11.

```
In [ ]:
1 from keras.models import load_model
2
3 model = load_model('filepath.h5')
```

***Figure A-11.*** *Loading a model given some file path*

Now that we've covered the basics of model construction and operation, let's move on to the parts that constitute the models themselves: **layers**.

# Layers

## Input Layer

`keras.layers.Input()`

This is the input layer of the entire model, and it has several parameters:

- **shape**: This is the shape tuple of integers that tells the layer what shape to expect. For example, if you pass in shape=(input_shape) and input_shape is (31, 1), you're telling the model to expect entries that each have a dimension (31, 1).

- **batch_shape**: This is also a shape tuple of integers that includes the batch size. Passing in batch_shape = (input_shape), where input_shape is (100, 31, 1), tells the model to expect batches of 100 31x1 dimensional entries. Passing in an input_shape of (None, 31, 1) tells the model that the number of batches can be some arbitrary number.
- **name**: (Optional) A string name for the layer. It must be unique, and if nothing is passed in, some name is autogenerated.
- **dtype**: The data type that the layer should expect the input data to have, specified as a string. It can be something like 'int32', 'float32', etc.
- **sparse**: A Boolean that tells the layer whether or not the placeholder that the layer creates is sparse.
- **tensor**: (Optional) A tensor to pass into the layer to serve as the placeholder for input. If something is passed in, then Keras will not automatically create some placeholder tensor.

## Dense Layer

`keras.layers.Dense()`

This is a neural network layer comprised of densely-connected neurons. Basically, every node in this layer is fully connected with the previous and next layers if there are any.

Here are the parameters:

- **units**: The number of neurons in this layer. This also factors into the dimension of the output space.
- **activation**: The activation function to use for this layer.
- **use_bias**: A Boolean for whether or not to use a bias vector in this layer.
- **kernel_initializer**: An initializer for the weight matrix. For more information, check out the **Initializers** section.
- **bias_initializer**: Similar to the kernel_initializer, but for the bias.

- **kernel_regularizer**: A regularizer function that's been applied to the weight matrix. For more information, check out the **Regularizers** section.
- **bias_regularizer**: Regularizer function applied to the bias.
- **activity_regularizer**: Regularizer function applied to the output of the layer.
- **kernel_constraint**: A constraint function applied to the weights. For more information, check out the **Constraints** section.
- **bias_constraint**: A constraint function applied to the bias.

For a better idea of what a dense layer is, check out Figure A-12.

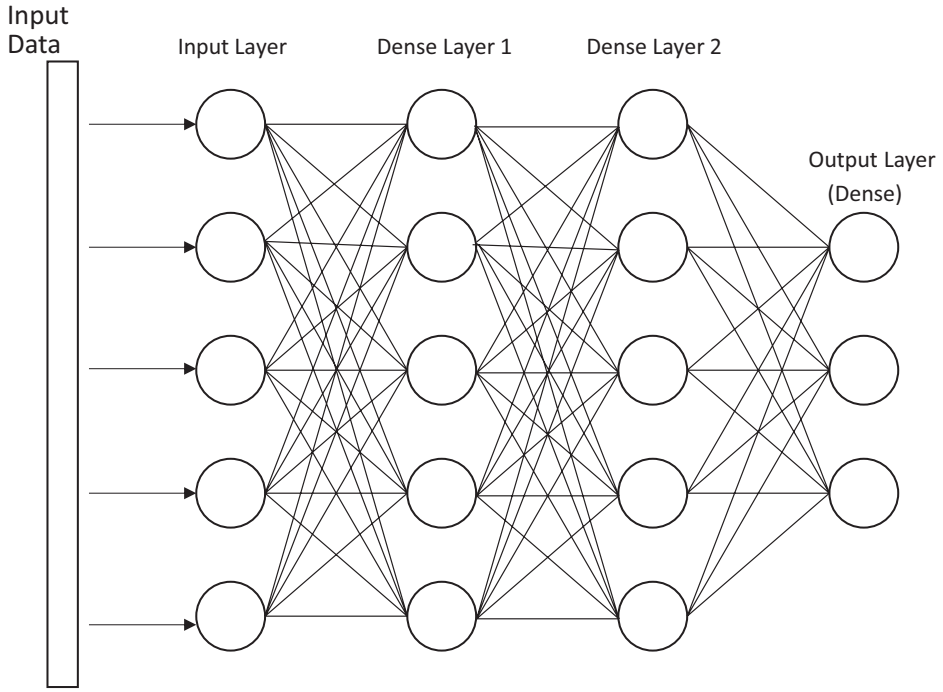



***Figure A-12.*** *Dense layers in an artificial neural network*

## Activation

`keras.layers.Activation()`

This layer applies an activation function to the input. Here is the argument:

- **activation**: Pass in either the activation function (see the **Activations** section) or some Theanos or TensorFlow operation.

To understand what an activation function is, Figure A-13 shows what each artificial neuron looks like.

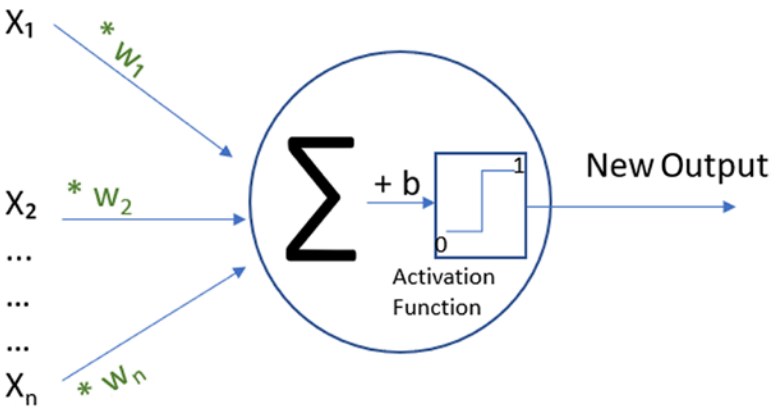



***Figure A-13.*** *The activation function is applied to the output of the function the node carries out on the input*

The activation passes in the output from the input ∗ weights + bias and passes it into the activation function. If there is no activation function, then that input just gets passed along as the output.

## Dropout

`keras.layers.Dropout()`

What the dropout layer does is take some float **f** proportion of nodes in the preceding layer and "deactivates" them, meaning they don't connect to the next layer. This can help combat overfitting on the training data.

Here are the parameters:

- **rate**: A float value between 0 and 1 that indicates the proportion of input units to drop.
- **noise_shape**: A 1D integer binary tensor that is multiplied with the input to determine what units are turned on or off. Instead of randomly selecting values using **rate**, you can pass in your own dropout mask to use in the dropout layer.
- **seed**: An integer to use as a random seed.

## Flatten

```
keras.layers.Flatten()
```

This layer takes all of the inputs and flattens them into a single dimension.

Images can have three channels if they're color images. They can be RGB (red, green, blue), BGR (blue, green, red), HSV (hue, saturation, value), etc., so the dimensions of these images are actually (height, width, channels) if it's formatted channels last or (channels, height, width) if it's formatted channels first. To preserve this formatting, there is a parameter you can pass in to the flatten layer:

- **data_format**: A string that's either 'channels_first' or 'channels_last'. This tells the flattening layer how to format the flattened output to preserve this formatting.

To get a better idea of how the layer flattens the input, check out the summary in Figure A-14 of a convolutional neural network.

```
In [5]:
import keras
from keras.models import Model
from keras.layers import Input, Dense, Convolution2D, Flatten

input_layer = Input(shape=(32, 32, 3))

conv_1 = Convolution2D(128, kernel_size=2, padding='same', activation='relu')(input_layer)

conv_2 = Convolution2D(128, kernel_size=2, padding='same', activation='relu')(conv_1)

flat_1 = Flatten()(conv_2)

classes = Dense(10, activation='softmax')(flat_1)

conv_net = Model(input_layer, classes)

conv_net.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_5 (InputLayer)         (None, 32, 32, 3)         0
_________________________________________________________________
conv2d_6 (Conv2D)            (None, 32, 32, 128)       1664
_________________________________________________________________
conv2d_7 (Conv2D)            (None, 32, 32, 128)       65664
_________________________________________________________________
flatten_2 (Flatten)          (None, 131072)            0
_________________________________________________________________
dense_2 (Dense)              (None, 10)                1310730
=================================================================
Total params: 1,378,058
Trainable params: 1,378,058
Non-trainable params: 0
_________________________________________________________________
```

***Figure A-14.*** *Notice how the flattening layer reduces the dimensionality of its input*

## Spatial Dropout 1D

`keras.layers.SpatialDropout1D()`

This function drops entire 1D feature maps instead of neuron elements, but otherwise has the same functionality as the regular dropout function. In earlier convolutional layers, the feature maps tend to be strongly correlated, so regular dropout functions won't help much with regularization in that case. Spatial dropout helps address this and also helps improve independence between the feature maps themselves.

The function takes one parameter:

- **rate**: A float between 0 and 1 that determines the proportion of input units to drop.

## Spatial Dropout 2D

`keras.layers.SpatialDropout2D()`

This function is similar to the spatial dropout 1D function, except it works on 2D feature maps. Images can have three channels if they're color images. They can be RGB (red, green, blue), BGR (blue, green, red), HSV (hue, saturation, value), etc., so the dimensions of these images are actually (height, width, channels) if it's formatted channels last or (channels, height, width) if it's formatted channels first.

This function takes one additional parameter compared to `SpatialDropout1D()`:

- **rate**: A float between 0 and 1 that determines the proportion of input units to drop.
- **data_format**: 'channels_first' or 'channels_last'. This tells the flattening layer how to format the flattened output to preserve the formatting of channels first or channels last.

## Conv1D

`keras.layers.Conv1D()`

Check out Chapter 7 for a detailed explanation on how one-dimensional convolutions work.

This layer is a one-dimensional (or temporal) convolutional layer. It basically passes a filter over the one-dimensional input and multiplies the values element-wise to create the output feature map.

These are the parameters that the function takes:

- **filters**: An integer value that determines the dimensionality of the output space. In other words, this is also the number of filters in the convolution.
- **kernel_size**: An integer (or tuple/list of a single integer) that specifies the length of the filter/kernel that is used in the 1D convolution.
- **strides**: An integer (or tuple/list of a single integer) that tells the layer how many data entries to shift by after one element-wise multiplication of the filter and the input data. Note: A stride value != 1 isn't compatible if the dilation_rate != 1.

- **padding:** 'valid', 'causal', or 'same'. 'valid' doesn't zero pad the output. 'same' zero pads the output so that it's the same length as the input. 'causal' padding generates causal, dilated convolutions. For an explanation on what 'causal' padding is, refer to Chapter 7.
- **data_format**: 'channels_first' or 'channels_last'. This tells the flattening layer how to format the flattened output to preserve the formatting of channels first or channels last. 'channels_first' has the format (`batch, features, steps`), and 'channels_last' has the format (`batch, steps, features`).
- **dilation_rate**: An integer (or tuple/list of a single integer) serves as the dilation rate for this dilated convolutional layer. For an explanation of how this works, refer to Chapter 7.
- **activation**: Passes in either the activation function (see the **Activations** section) or some Theanos or TensorFlow operation. If nothing is specified, the data is passed along unaltered after the convolutional process.
- **use_bias**: A Boolean for whether or not to use a bias vector in this layer.
- **kernel_initializer**: An initializer for the weight matrix. For more information, check out the **Initializers** section.
- **bias_initializer**: Similar to the kernel_initializer, but for the bias.
- **kernel_regularizer**: A regularizer function that's been applied to the weight matrix. For more information, check out the **Regularizers** section.
- **bias_regularizer**: A regularizer function applied to the bias.
- **activity_regularizer**: A regularizer function applied to the output of the layer.
- **kernel_constraint**: A constraint function applied to the weights. For more information, check out the **Constraints** section.
- **bias_constraint**: A constraint function applied to the bias.

## Conv2D

`keras.layers.Conv1D()`

Check out Chapter 3 for a detailed explanation on how the 2D convolutional layer works.

This layer is a two-dimensional convolutional layer. It basically passes a 2D filter over the input and multiplies the values element-wise to create the output feature map.

These are the parameters that the function takes:

- **filters**: An integer value that determines the dimensionality of the output space. In other words, this is also the number of filters in the convolution.
- **kernel_size**: An integer (or tuple/list of two integers) that specifies the height and width of the filter/kernel that is used in the 2D convolution.
- **strides**: An integer (or tuple/list of two integers, one for height and one for width, respectively) that tells the layer how many data entries to shift by after one element-wise multiplication of the filter and the input data. Note: A stride value != 1 isn't compatible if the dilation_rate != 1.
- **padding**: 'valid' or 'same'. 'valid' doesn't zero pad the output. 'same' zero pads the output so that it's the same length as the input.
- **data_format**: 'channels_first' or 'channels_last'. This tells the flattening layer how to format the flattened output to preserve the formatting of channels first or channels last.
- **dilation_rate**: An integer (or tuple/list of a two integers) serves as the dilation rate for this dilated convolutional layer. For an explanation of how this works, refer to Chapter 7.
- **activation**: Passes in either the activation function (see the **Activations** section) or some Theanos or TensorFlow operation. If nothing is specified, the data is passed along unaltered after the convolutional process.
- **use_bias**: A Boolean for whether or not to use a bias vector in this layer.

- **kernel_initializer**: An initializer for the weight matrix. For more information, check out the **Initializers** section.
- **bias_initializer**: Similar to the kernel_initializer, but for the bias.
- **kernel_regularizer**: A regularizer function that's been applied to the weight matrix. For more information, check out the **Regularizers** section.
- **bias_regularizer**: A regularizer function applied to the bias.
- **activity_regularizer**: A regularizer function applied to the output of the layer.
- **kernel_constraint**: A constraint function applied to the weights. For more information, check out the **Constraints** section.
- **bias_constraint**: A constraint function applied to the bias.

## UpSampling 1D

`keras.layers.UpSampling1D()`

For a detailed explanation on how upsampling works, refer to Chapter 7.

This layer essentially repeats the data **n** times with respect to time (where **n** is the parameter passed in):

- **size**: An integer **n** that specifies how many times to repeat each data entry with respect to time. The order of time is preserved, so each element is repeated **n** times according to its time entry.

## UpSampling 2D

`keras.layers.UpSampling2D()`

Similar to UpSampling1D(), but for 2D inputs. The rows and columns are repeated **n** times according to size[0] and size[1].

This is the list of parameters:

- **size**: An integer or tuple of two integers. The integer is the upsampling factor for both rows and columns, and the tuple lets you specify the upsampling factor for rows and for columns individually.

- **data_format**: 'channels_first' or 'channels_last'. This tells the flattening layer how to format the flattened output to preserve the formatting of channels first or channels last.
- **interpolation**: 'nearest' or 'bilinear'. CNTK does not support 'bilinear' yet, and Theanos only supports size=(2,2). 'nearest' and 'bilinear' are interpolation techniques used in image processing.

## ZeroPadding1D

`keras.layers.ZeroPadding1D()`

Depending on the input, pads the input sequence with zeroes on both sides or either a zero on the left side or a zero on the right side of the input sequence.

This is the list of parameters:

- **padding**: An integer, a tuple of two integers, or a dictionary. The integer is a number that tells the layer how many zeroes to add on both the left and right side. An input of **1** adds a zero on both the left and right side. The tuple is formatted as (`left_pad, right_pad`), so you can pass in (0, 1) to tell it to add no zeroes on the left side and add one zero on the right side.

## ZeroPadding2D

`keras.layers.ZeroPadding2D()`

Depending on the input, it pads the input sequence with a row and columns of zeroes at the top, left, right, and bottom of the image tensor.

This is the list of parameters:

- **padding**: An integer, a tuple of two integers, a tuple of two tuples with two integers each. The integer tells it to add **n** rows of zeroes on the top and bottom of the image tensor, and **n** columns of zeroes. The tuple of two integers is formatted as (`symmetric_height_pad, symmetric_width_pad`), so you can tell the layer to add **m** rows of zeroes and **n** columns of zeroes to each side, respectively, if you pass in a tuple (**m, n**). Finally, the tuple of two tuples is formatted as ((`top_pad, bottom_pad`), (`left_pad, right_pad`)), so you can customize even more how you want the layer to add rows or columns of zeroes.

- **data_format**: 'channels_first' or 'channels_last'. This tells the flattening layer how to format the flattened output to preserve the formatting of channels first or channels last.

## MaxPooling1D

`keras.layers.MaxPooling1D()`

It applies max pooling on a 1D input. To get a better idea of how max pooling works, check out Chapter 3. Max pooling in 1D is similar to max pooling in 2D, except the sliding window only works in one dimension, going from left to right.

This is the list of parameters:

- **pool_size**: An integer value. If an integer **n** is given, then the window size of the pooling layer is **1xn.** These are also the factors to downscale by, so if an integer **n** is passed in, the dimensions for both height and width are downscaled by that factor.
- **strides**: An integer or None. By default, the stride is set to **pool_size**. If you pass in an integer, the pooling window moves by integer **n** amount after completing its pooling operation on a set of entries.
- **padding**: 'valid' or 'same'. 'valid' means there's no zero padding, and 'same' pads the output sequence with zeroes so that it matches the dimensions of the input sequence.
- **data_format**: 'channels_first' or 'channels_last'. This tells the flattening layer how to format the flattened output to preserve the formatting of channels first or channels last. 'channels_first' has the format (`batch, features, steps`), and 'channels_last' has the format (`batch, steps, features`).

## MaxPooling2D

`keras.layers.MaxPooling2D()`

It applies max pooling on a 2D input. To get a better idea of how max pooling works, check out Chapter 3.

This is the list of parameters:

- **pool_size**: An integer that dictates the size of the pooling window. An integer of **n** makes the pooling window size **n**, meaning it sifts through **n** entries at a time and selects the maximum value to pass on to the output.
- **strides**: An integer or None. By default, the stride is set to **pool_size**. If you pass in an integer, the pooling window moves by integer **n** amount after completing its pooling operation on a set of entries. It is also a factor that determines how much to downscale the dimensions by, as a parameter **n** will reduce the dimensions by a factor **n**.
- **padding**: 'valid' or 'same.' 'valid' means there's no zero padding, and 'same' pads the output sequence with zeroes so that it matches the dimensions of the input sequence.
- **data_format**: 'channels_first' or 'channels_last.' This tells the flattening layer how to format the flattened output to preserve the formatting of channels first or channels last.

# Loss Functions

In the examples, y_true is the true label and y_pred is the predicted label.

## Mean Squared Error

```
keras.losses.mean_squared_error(y_true, y_pred)
```

If you have questions on the notation for this equation, refer to Chapter 3. See the equation in Figure A-15.

$$J(\theta) = \frac{1}{n}\sum_{i=1}^{n}\left(h_\theta\left(x^i\right) - y^i\right)^2$$

***Figure A-15.*** *The equation for mean squared error*

Given input $\theta$, the weights, the formula finds the average difference squared between the predicted value and the actual value. The parameter $h_\theta$ represents the model with the weight parameter $\theta$ passed in, so $h_\theta(x^i)$ gives the predicted value for $x^i$ with model's weights $\theta$. The parameter $y^i$ represents the actual prediction for the data point at index i. Lastly, there are n entries in total.

This loss metric can be used in autoencoders to help evaluate the difference between the reconstructed output and the original. In the case of anomaly detection, this metric can be used to separate the anomalies from the normal data points, since anomalies have a higher reconstruction error.

## Categorical Cross Entropy

```
keras.losses.categorical_crossentropy(y_true, y_pred)
```

See the equation in Figure A-16.

$$J(\theta) = -\frac{1}{n}\sum_{i=0}^{n} y_i * log(h_\theta(x_i)) + (1 - y_i) * \log(1 - h_\theta(x_i))$$

***Figure A-16.*** *The equation for categorical cross entropy*

In this case, **n** is the number of samples in the whole data set. The parameter $h_\theta$ represents the model with the weight parameter $\theta$ passed in, so $h_\theta(x_i)$ gives the predicted value for $x_i$ with model's weights $\theta$. Finally, $y_i$ represents the true label for data point at index i. The data needs to be regularized to be between 0 and 1, so for categorical cross entropy, it must be piped through a softmax activation layer. The categorical cross entropy loss is also called **softmax loss**.

Equivalently, you can write the previous equation as shown in Figure A-17.

$$J(\theta) = -\frac{1}{n}\sum_{i=0}^{n}\sum_{j=0}^{m} y_{ij} * \log(h_\theta(x_{ij})$$

***Figure A-17.*** *Another way to write the equation for categorical cross entropy*

In this case, **m** is the number of classes.

The categorical cross entropy loss is a commonly used metric in classification tasks, especially in computer vision with convolutional neural networks. **Binary cross entropy** is a special case of categorical cross entropy where the number of classes **m** is two.

## Sparse Categorical Cross Entropy

`keras.losses.sparse_categorical_crossentropy(y_true, y_pred)`

Sparse categorical cross entropy is basically the same as categorical cross entropy, but the distinction between them is in how their true labels are formatted. For categorical cross entropy, the labels are **one-hot encoded**. For an example of this, refer to Figure A-18, if you had your y_train formatted originally as the following, with six maximum classes.

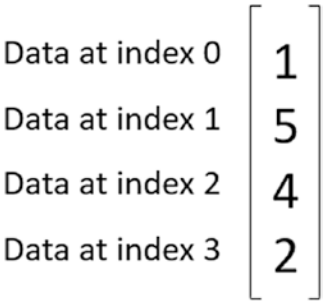



***Figure A-18.*** *An example of how* `y_train` *can be formatted. The value in each index is the class value that corresponds to the value at that index in* `x_train`

You can call `keras.utils.to_categorical(y_train, n_classes)` with n_classes as 6 to convert y_train to that shown in Figure A-19.

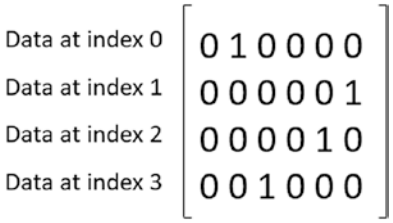



***Figure A-19.*** *The* `y_train` *in Figure A-18 is converted into a one-hot encoded format*

So now your `y_train` looks like Figure A-20.

```
In [7]:  1 y_train = [1, 5, 4, 2]
         2
         3 keras.utils.to_categorical(y_train, 6)

Out[7]: array([[0., 1., 0., 0., 0., 0.],
               [0., 0., 0., 0., 0., 1.],
               [0., 0., 0., 0., 1., 0.],
               [0., 0., 1., 0., 0., 0.]])
```

***Figure A-20.*** *Converting* `y_train` *into a one-hot encoded format in Jupyter*

This type of truth label formatting (**one-hot encoding)** is what categorical cross entropy uses. For sparse categorical cross entropy, it suffices to simply pass in the information in Figure A-21.

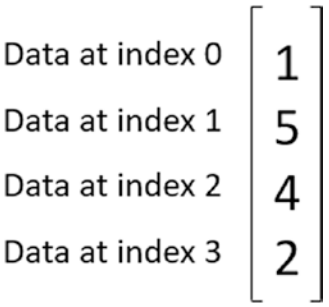



***Figure A-21.*** *The* `y_train` *to pass in for sparse categorical cross entropy*

Or the code shown in Figure A-22.

```
In [7]:  1 y_train = [1, 5, 4, 2]
         2
```

***Figure A-22.*** *An example of* `y_train` *in the code that can be passed in if sparse categorical cross entropy is the metric*

# Metrics

## Binary Accuracy

`keras.metrics.binary_accuracy(y_true, y_pred)`

To use this function, the 'accuracy' must be a metric that's passed into the `model.compile()` function, and binary cross entropy must be the loss function.

Essentially, the function finds the number of instances where the true class label matches the rounded prediction label and finds the mean of the result (which is the same thing as dividing the total number of correct matches by the total number of samples).

The predicted values are rounded since as the neural network is trained more and more, the output values tend to change so that the predicted value is something really close to one, and the rest of the value are something really close to zero. In order to match the predicted values to the original truth labels (which are all integers), you can simply round the predicted values.

In the official Keras documentation on GitHub, this function is defined as shown in Figure A-23.

```
6  def binary_accuracy(y_true, y_pred):
7      '''Calculates the mean accuracy rate across all predictions for binary
8      classification problems.
9      '''
10     return K.mean(K.equal(y_true, K.round(y_pred)))
```

***Figure A-23.*** *The code definition in the Keras GitHub page of binary accuracy*

## Categorical Accuracy

`keras.metrics.categorical_accuracy(y_true, y_pred)`

Since most problems tend to involve categorical cross entropy (implying more than two classes in the data set), this tends to be the default accuracy metric when 'accuracy' is passed into the `model.compile()` function.

Instead of finding all of the instances where the true labels and rounded predictions match, categorical accuracy finds all of the instances where the true labels and predictions have a maximum value in the same spot.

Recall that for categorical cross entropy, the labels are one-hot encoded. Therefore, the truth labels only have one maximum per entry, along with the predictions (though again, one value will be really close to one while the others are really close to zero). What categorical accuracy does is check if the maximum value in the entry is in the same position for both y_true and for y_pred.

Once it's found all those instances, it finds the mean of the result, leading to an accuracy value.

Essentially, it's a similar equation to the one for binary accuracy, but with a different condition regarding y_true and y_pred.

The function is defined by Keras as shown in Figure A-24.

```
def categorical_accuracy(y_true, y_pred):
    '''Calculates the mean accuracy rate across all predictions for
    multiclass classification problems.
    '''
    return K.mean(K.equal(K.argmax(y_true, axis=-1),
                  K.argmax(y_pred, axis=-1)))

```

***Figure A-24.*** *The code definition of categorical accuracy as seen in the Keras GitHub page*

Of course, there are many more metrics that are available on the Keras documentation, and you can even define **custom metrics**. To do that, just simply define a function that takes in y_true and y_pred, and call that function name in your metrics, as shown in Figure A-25.

```
In [ ]:
import keras.backend as K

def custom_metric(y_true, y_pred):
    matches = K.equal(y_true, K.round(y_pred))
    score = K.mean(matches)
    return  score

model.compile(optimizer='optimizer',
              loss='loss_function',
              metrics=['accuracy', custom_metric])
```

***Figure A-25.*** *Code to define a custom metric and use that for the model*

In this example, you simply rewrite the binary accuracy metric in several lines and return the score. You can actually condense this function to just one line like in the actual implementation seen above, but this is just an example to showcase a custom metric.

# Optimizers

## SGD

`keras.optimizers.SGD()`

This is the **stochastic gradient descent** optimizer, a type of algorithm that aids in the backpropagation process by adjust the weights. It is commonly used as a training algorithm in a variety of machine learning applications, including neural networks.

The optimizer has several parameters:

- **lr:** Some float value where the learning rate **lr** >= 0. The learning rate is a hyperparameter that determines how big of a step to take when optimizing the loss function.
- **momentum**: Some float value where the momentum **m** >= 0. This parameter helps accelerate the optimization steps in the direction of the optimization, and helps reduce oscillations when the local minimum is overshot (refer to Chapter 3 to refresh your understanding on how a loss function is optimized).
- **decay**: Some float value where the decay **d** >= 0. Helps determine how much the learning rate decays by after each update (so that as the local minimum is approached, or after some number of training iterations, the learning rate decreases so smaller step sizes are taken. Big learning rates means the local minimum might be overshot more easily).
- **nesterov**: A Boolean value to determine whether or not to apply Nesterov momentum. Nesterov momentum is a variation of momentum where the gradient is computed not from the current position, but from a position that takes into account the momentum. This is because the gradient always points in the right direction, but the momentum might carry the position too far forward and overshoot. Since it doesn't use the current position but instead some intermediate position that takes into account momentum, the gradient from that position can help correct the current course so that the momentum doesn't carry the new weights too far forward.

It essentially helps for more accurate weight updates and helps converge faster.

## Adam

```
keras.optimizers.Adam()
```

The Adam optimizer is an algorithm that extends upon SGD, and has grown quite popular in deep learning applications in computer vision and in natural language processing.

These are the parameters for the algorithm:

- **lr**: Some float value where the learning rate **lr** >= 0. The learning rate is a hyperparameter that determines how big of a step to take when optimizing the loss function. The paper describes good results with a value of 0.001 (the paper refers to the learning rate as **alpha**).
- **beta_1**: Some float value where 0 < **beta_1** < 1. This is usually some value close to 1, but the paper describes good results with a value of 0.9.
- **beta_2**: Some float value where 0 < **beta_2** < 1. This is usually some value close to 1, but the paper describes good results with a value of 0.999.
- **epsilon**: Some float value where epsilon **e** >= 0. If None, then it defaults to `K.epsilon()`. Epsilon is some small number, described as 10E-8 in the paper, to help prevent division by 0.
- **decay**: Some float value where the decay **d** >= 0. Helps determine how much the learning rate decays by after each update (so that as the local minimum is approached, or after some number of training iterations, the learning rate decreases so smaller step sizes are taken. Big learning rates means the local minimum might be overshot more easily).
- **amsgrad**: A Boolean on whether or not to apply the AMSGrad version of this algorithm. For more details on the implementation of this algorithm, check out "On the Convergence of Adam and Beyond."

## RMSprop

`keras.optimizers.RMSprop()`

RMSprop is a good algorithm for recurrent neural networks. RMSprop is a gradient-based optimization technique developed to help address the problem of gradients becoming too large or too small. RMSprop helps combat this problem by normalizing the gradient itself using the average of the squared gradients. In Chapter 7, it's explained that one of the problems with RNNs is the vanishing/exploding gradient problem, leading to the development of LSTMs and GRU networks. And so it's of no surprise that RMSprop pairs well with recurrent neural networks.

Besides the learning rate, it's recommended to leave the rest of the algorithms in their default settings. With that in mind, here are the parameters for this optimizer:

- **lr**: Some float value where the learning rate **lr** >= 0. The learning rate is a hyperparameter that determines how big of a step to take when optimizing the loss function.
- **rho**: Some float value where **rho** >= 0. Rho is a parameter that helps calculate the exponentially weighted average over the gradients squared.
- **epsilon**: Some float value where epsilon **e** >= 0. If None, then it defaults to `K.epsilon()`. Epsilon is a very small number that helps prevent division by 0 and to help prevent the gradients from blowing up in RMSprop.
- **decay**: Some float value where the decay **d** >= 0. Helps determine how much the learning rate decays by after each update (so that as the local minimum is approached, or after some number of training iterations, the learning rate decreases so smaller step sizes are taken. Big learning rates means the local minimum might be overshot more easily).

# Activations

You can pass in something like '`activation_function`' for the activation parameter in a layer, or the full function, `keras.activations.activation_function()`, if you want to customize it more. Otherwise, the default initialized activation function is used in the layer.

## Softmax

`keras.activations.softmax()`

This performs a softmax activation on the input **x** and on the given axis. The two parameters are

- **x**: The input tensor
- **axis**: The axis that you want to use softmax normalization on. By default, it is set to -1.

The general formula for softmax is shown in Figure A-26 (K is the number of samples).

$$\sigma(x)_i = \frac{e^{x_i}}{\sum_{j=1}^{K} e^{x_j}} \quad \text{For } i = 1,\ldots,K \text{ and } x = (x_1, \ldots, x_k) \in R^k$$

***Figure A-26.*** *The general formula for softmax*

## ReLU

`keras.activations.relu()`

ReLU, or "Rectified Linear Unit", performs a simple activation based on the function shown in Figure A-27.

$$f(x) = \max(0, x)$$

***Figure A-27.*** *This is the general ReLU formula*

The parameters are as follows:

- **x**: The input tensor
- **alpha**: A float that determines the slope of the negative part. Set to zero by default.
- **max_value**: A float value that represents the upper threshold, and is set to None by default.
- **threshold**: A float value set to 0.0 by default that's the lower threshold.

If **max_value** is set, then you get the equation shown in Figure A-28.

$$f(x) = \text{max\_value} \quad \text{for } x \geq \text{max\_value}$$

***Figure A-28.*** *The ReLU formula if max_value is set*

If **threshold** is also set, then you get the equation shown in Figure A-29.

$$f(x) = x \quad \text{for threshold} \leq x < \text{max\_value}$$

***Figure A-29.*** *The ReLU formula if threshold is also set*

Otherwise you get the equation shown in Figure A-30.

$$f(x) = alpha * (x - threshold)$$

***Figure A-30.*** *The formula for ReLU if alpha and threshold are set*

For an example of what the base ReLU function does, refer to Figure A-31.

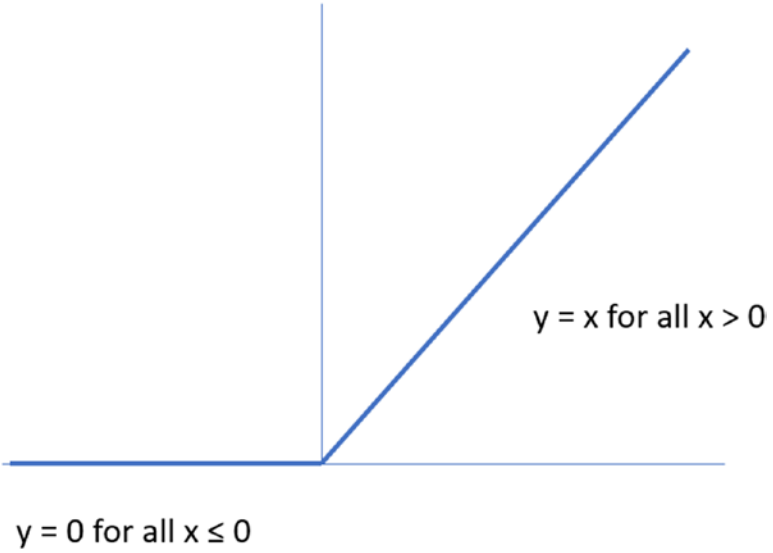



***Figure A-31.*** *The graph for a basic ReLU function*

## Sigmoid

```
keras.activations.sigmoid(x)
```

This is a simple activation function to call, as there are no parameters other than the input tensor **x**.

The sigmoid function does have its uses, primarily because it forces the input to be between 0 and 1, but it is prone to the vanishing gradient problem, and so it is seldom used in hidden layers.

To get an idea of what the equation is like when graphed, refer to Figure A-32.

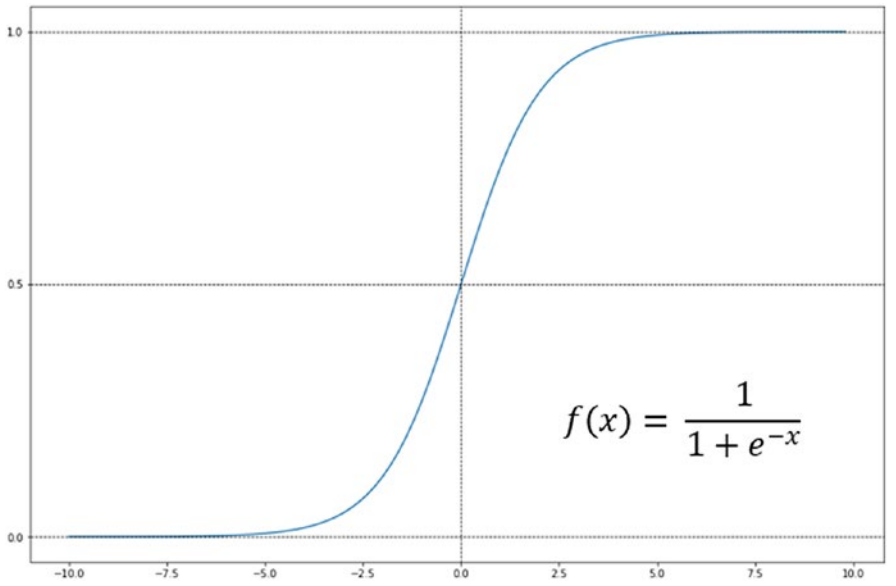



***Figure A-32.*** *The graph of a sigmoid function*

# Callbacks

## ModelCheckpoint

`keras.callbacks.ModelCheckpoint()`

ModelCheckpoint is basically a function that saves the model every epoch (unless otherwise directed via parameters). How it does so can be configured by the set of parameters associated with `ModelCheckpoint()`:

- **filepath**: The path where you want to save the model file. Typing just "model_name.h5" saves it in the same directory.
- **monitor**: The quantity that you want the model to monitor. By default, it's set to "val_loss".
- **verbose**: Sets verbosity to 0 or 1. It's set to 0 by default.
- **save_best_only**: If set to true, then the model with the best performance according to the quantity monitored will be saved.
- **save_weights_only**: If set to True, then only the weights will be saved. Essentially, if True, `model.save_weights(filepath)`; else, `model.save(filepath)`.

- **mode**: Choose between auto, min, or max. If save_best_only is True, then you should pick a choice that would suit the monitored quantity best. If you chose val_acc for monitor, then you want to pick max for mode, and if you choose val_loss for monitor, pick min for mode.
- **period**: How many epochs there are between each checkpoint.

## TensorBoard

```
keras.callbacks.TensorBoard()
```

TensorBoard is a visualization tool that comes with TensorFlow. It helps you see in detail what's going on as your model trains.

To launch TensorBoard, type this into the command prompt:

```
tensorboard --logdir=/full_path_to_your_logs
```

```
keras.callbacks.TensorBoard(log_dir='./logs', histogram_freq=0, batch_
size=32, write_graph=True, write_grads=False, write_images=False,
embeddings_freq=0, embeddings_layer_names=None, embeddings_metadata=None,
embeddings_data=None, update_freq='epoch')
```

With that, here is the list of parameters:

- **log_dir**: The path to the directory where you want the model to save the log files. This is the same directory you pass as an argument in the command prompt. It is './logs' by default.
- **histogram_freq**: The frequency (in epochs) that you want the activation and weight histograms to be computed for the model's layers. Set to 0 by default, which means it won't compute histograms. To visualize these histograms, validation_data (or validation_split) must be passed in.
- **batch_size**: The size of each batch of inputs to pass into the network to compute histograms from. Set to 32 by default.
- **write_graph**: Whether or not to allow the graph to be visualized in TensorBoard. Set to True by default. Note: When set to True, the log files can become large.

- **write_grads**: Whether or not to allow TensorBoard to visualize the gradient histograms. Set to False by default, and also needs histogram_freq to be a value greater than 0.
- **write_images**: Whether or not to visualize the model weights as an image in TensorBoard. Set to False by default.
- **embeddings_freq**: The frequency, in epochs, to save selected embedding layers. Set to 0 by default, which means that the embeddings won't be computed. To visualize data in TensorBoard's Embedding tab, pass in the data as embeddings_data.
- **embeddings_layer_names**: The list of names of layers for TensorBoard to track. If None or an empty list, then all of the layers will be watched. Set to None by default.
- **embeddings_metadata**: A dictionary that maps layer names to the corresponding file names where the metadata for this embedding layer is saved. Set to None by default. If the same metadata file is used for all of the embedding layers, then a string can be passed.
- **embeddings_data**: The data to be embedded at the layers specified in embeddings_layer_names. This is a Numpy array if the model expects a single input, and multiple Numpy arrays if the model has multiple inputs. Set to None by default.
- **update_freq**: A 'batch', 'epoch', or integer. 'batch' writes the losses and metrics to TensorBoard after each batch. 'epoch' is the same, except the losses and metrics are written to TensorBoard after each epoch. The integer tells it to write the metrics and losses to TensorBoard every integer **n** samples, where **n** is the integer passed in. Note: Writing to TensorBoard too frequently can slow down the training process.

With that being said, Figure A-33 shows an example of using TensorBoard as a callback when training a convolutional neural network on the MNIST data set.

```
tensorboard = keras.callbacks.TensorBoard(log_dir='./Graph',
histogram_freq=0,
          write_graph=True, write_images=True)


checkpoint =
keras.callbacks.ModelCheckpoint(filepath="keras_MNIST_CNN.h5",
                                verbose=0,
                                save_best_only=True)


model.fit(x_train, y_train,
          batch_size=batch_size,
          epochs=n_epochs,
          verbose=1,
          validation_data=(x_test, y_test),
          callbacks=[checkpoint, tensorboard])
```

***Figure A-33.*** *Code to define a TensorBoard callback and use that when training*

Once you execute that code, you will notice the training process will begin. At this point, enter the line

```
tensorboard --logdir=/full_path_to_your_logs
```

into your command prompt and press Enter. It should show you something like Figure A-34.

```
TensorBoard 1.10.0 at http://MSI:6006 (Press CTRL+C to quit)
```

***Figure A-34.*** *You should see something like this after executing the above line in command prompt. It should tell you where to go to access TensorBoard, which is http://MSI:6006 in this case*

Simply follow that link and you should see the screen shown in Figure A-35.

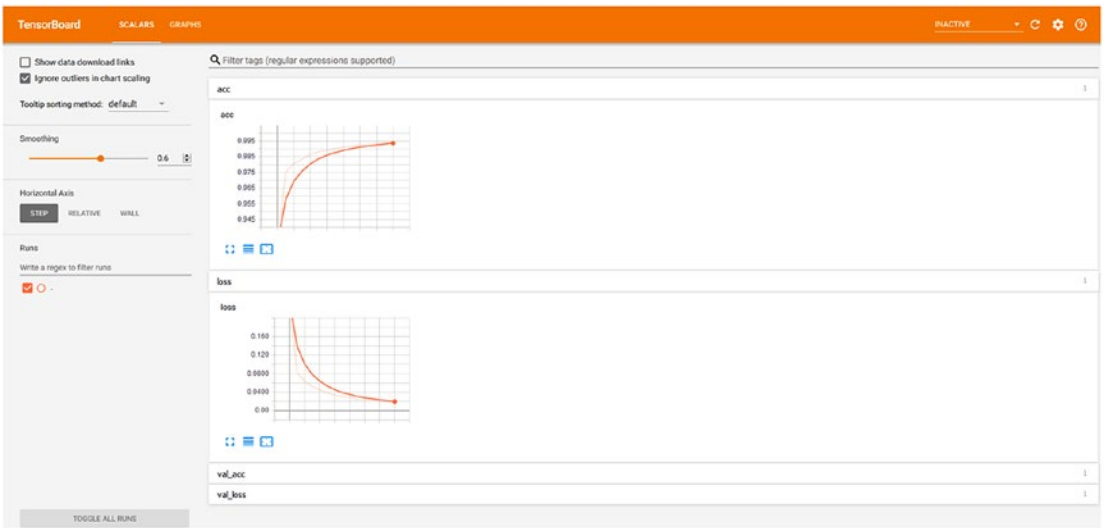



***Figure A-35.*** *The general page that appears when you launch TensorBoard*

From here, you can see graphs for the metrics accuracy and loss. You can expand the other two metrics, val_acc and val_loss, to view those graphs as well (see Figure A-36).


***Figure A-36.*** *Graphs for val_acc and val_loss*

As for the individual graphs, you can expand them out by pressing the leftmost button below the graph, and you can view data on the graph as you move your mouse across it, as seen in Figure A-37.

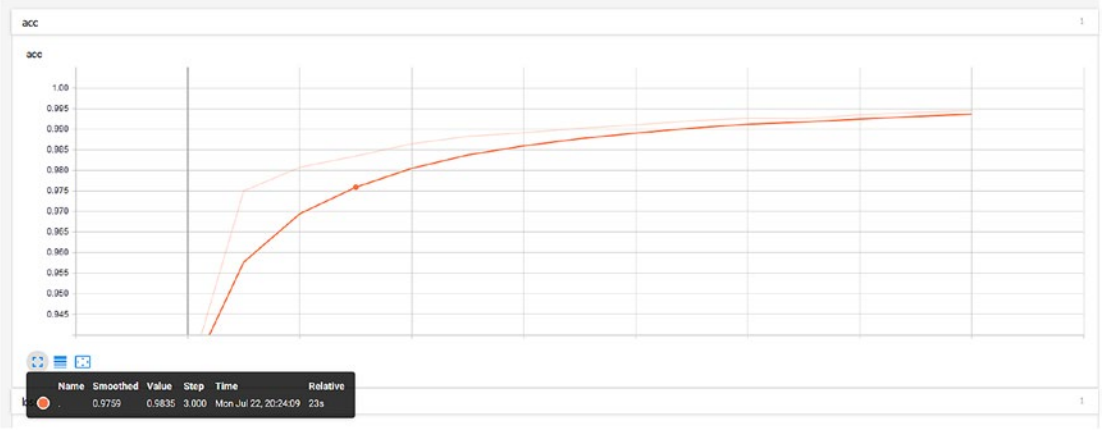



***Figure A-37.*** *The result of pressing the leftmost button underneath the graph. Doing so expands the graph, and regardless of whether the graph is expanded or not, you can point your mouse cursor at any point along the graph to get more details about that point*

You can also view a graph of the entire model by pressing the Graphs tab, as shown in Figure A-38.


***Figure A-38.*** *There are two tabs. You started on the tab named SCALARS. Press GRAPHS to switch the tab*

Doing so will result in a graph similar to the one shown in Figure A-39.

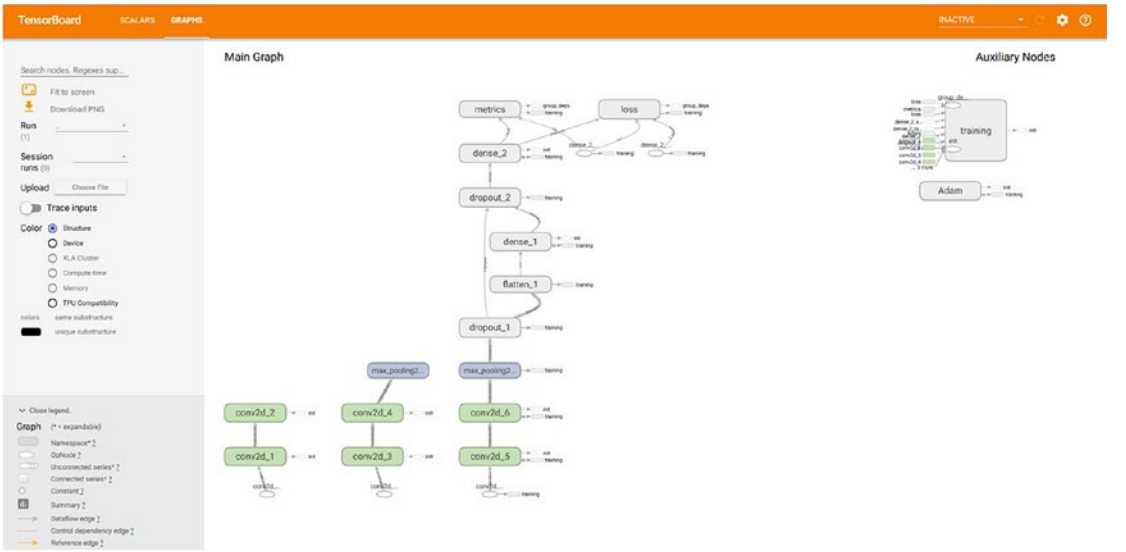



***Figure A-39.*** *The result of clicking on the GRAPHS tab*

There are definitely more features and functionality that TensorBoard offers, but the general idea is that you will be able to examine your models in a much better fashion.

## Back End (TensorFlow Operations)

You can also perform operations with TensorFlow (if it is the back end) through Keras by importing the back end. Below, we will demonstrate some basic functions, but keep in mind that TensorFlow has a vast variety of operations and functions.

You can use the back end to create custom layers, metrics, loss functions, etc., allowing for a much deeper level of customization. However, you must basically be knowledgeable in TensorFlow to accomplish all of this, since this is practically just using TensorFlow.

If you want the most customization possible, then using tf.keras along with TensorFlow is better, since tf.keras is wholly compatible with all of TensorFlow, and you'll have access to many more TensorFlow commands that you can't get with just the Keras back end.

Here are some of the commands you can execute using the back end (Figure A-40, Figure A-41, Figure A-42, Figure A-43).

```
In [125]:
1  import keras.backend as K
2
3  #Declaring a placeholder
4  a = K.placeholder(shape=(1,2,3)) # Equivalent to tf.placeholder()
5  print(a)
6
7  vals = [0, 1, 2, 3, 4, 5]
8  b = K.variable(value=vals) # Equivalent to tf.Variable()
9  print(b)

Tensor("Placeholder_62:0", shape=(1, 2, 3), dtype=float32)
<tf.Variable 'Variable_65:0' shape=(6,) dtype=float32_ref>
```

***Figure A-40.*** *Some TensorFlow operations such as defining placeholders and variables done through the Keras back end*

```
In [126]:
1  import keras.backend as K
2
3  c = K.placeholder(shape=(1, 2))
4  d = K.placeholder(shape=(2, 5))
5
6  print(K.dot(c, d))

Tensor("MatMul_13:0", shape=(1, 5), dtype=float32)
```

***Figure A-41.*** *Finding the dot product of two placeholder variables c and d using the Keras back end*

```
In [134]:
1  print(K.sum(c, axis=0))
2  print(K.sum(c, axis=1))

Tensor("Sum_7:0", shape=(2,), dtype=float32)
Tensor("Sum_8:0", shape=(1,), dtype=float32)
```

***Figure A-42.*** *Finding the sum of c along different axes using the Keras back end*

```
In [136]:
1  print(K.mean(c, axis=0))

Tensor("Mean_1:0", shape=(2,), dtype=float32)
```

***Figure A-43.*** *Finding the mean of c using the Keras back end*

Those are just some of the most basic functions available through the back end. The complete list of backend functions is available at https://keras.io/backend/.

## Summary

Keras is a great tool to help you easily get involved with creating, training, and testing deep learning models, and provides a great deal of functionality while abstracting away the complicated syntax that TensorFlow has. Keras by itself can be sufficient, but as the content gets more advanced, it's better to have the level of customization and flexibility that TensorFlow or PyTorch offers. Keras allows you to use a wide variety of functions through the back end, allowing you to write custom layers, custom models, metrics, loss functions, and so on, but for the most customization and flexibility in how you want your neural networks to be (especially if you want to make completely new types of neural networks), then either tf.keras + TensorFlow or PyTorch would be better suited for your needs.

# APPENDIX B

# Intro to PyTorch

In this appendix, you will be introduced to the PyTorch framework along with the functionality that it offers. PyTorch is more involved than Keras is, and it is a lower-level framework (meaning there's more syntax, and elements aren't abstracted away from you like in Keras).

Regarding the setup, we use

- Torch version 0.4.1 (PyTorch)
- CUDA version 9.0.176
- cuDNN version 7.3.0.29

## What Is PyTorch?

PyTorch is a deep learning library for Python, developed by artificial-intelligence researchers at Facebook and based on the Torch library. While PyTorch is also a low-level language like TensorFlow, it is easier to pick up because of the huge difference in syntax. TensorFlow has a much steeper learning curve, and you have to define a lot more elements than in PyTorch.

TensorFlow at the moment far surpasses PyTorch in how much community support it has, and this is primarily because PyTorch is a relatively new framework. Although you will find more resources for TensorFlow, more and more people are switching to PyTorch due to it being more intuitive while still offering practically the same functionality as TensorFlow (though TensorFlow does have some functions that PyTorch does not, you can easily implement those functions in PyTorch if you know what the logic is; an example of this is `arctanh` function).

In the end, it is mostly a matter of personal preference when deciding to use TensorFlow or PyTorch. Depending on the context of your work, one framework might be more suitable than the other.




S. Alla and S. K. Adari, *Beginning Anomaly Detection Using Python-Based Deep Learning*,
https://doi.org/10.1007/978-1-4842-5177-5

That being said, PyTorch might be easier to use for research purposes, considering that it is easier to prototype in due to the lessened burden from the syntax. On the other hand, TensorFlow has more resources and the advantage of having TensorBoard. It is also better suited for cross-platform compatibility, since a model can be trained in Python but deployed in Java, for example, allowing for better scalability. If loading and saving models is a priority, perhaps TensorFlow is more suitable. Again, it all comes down to personal preference, since there's usually a workaround for many of the problems that both frameworks might face.

# Using PyTorch

This section will be a bit different from the previous appendix. Here, we will demonstrate how some basic tensor operations are done, and then move on to illustrating how to use PyTorch by exploring PyTorch equivalent models of the temporal convolutional networks in Chapter 7.

First, let's begin by looking at some simple tensor operations. If you would like to know more about the framework itself and the functionality that it supports, check out the documentation at `https://pytorch.org/docs/0.4.1/index.html` and the code implementation at `https://github.com/pytorch/pytorch`.

Let's begin (see Figure B-1).

```
In [53]:
1 import torch
2 import torch.nn
3 import numpy as np
4
5 a = np.random.randint(0, 10, 5)
6 a = torch.tensor(a)
7 a

Out[53]: tensor([0, 3, 0, 9, 4], dtype=torch.int32)

In [54]:
1 b = torch.tensor(np.random.randint(0, 10, 5))
2 b

Out[54]: tensor([3, 9, 7, 4, 2], dtype=torch.int32)

In [55]:
1 c = torch.add(a, b)
2 c_summed = torch.sum(c)
3
4 c, c_summed

Out[55]: (tensor([ 3, 12,  7, 13,  6], dtype=torch.int32), tensor(41))

In [56]:
1 d = torch.tanh(b.float())
2 d

Out[56]: tensor([0.9951, 1.0000, 1.0000, 0.9993, 0.9640])

In [57]:
1 torch.mean(d)

Out[57]: tensor(0.9917)
```

***Figure B-1.*** *A series of tensor operations in PyTorch. The code shows the operation and the output shows the results after the operations were performed on the corresponding tensors*

With PyTorch, you can see that the data values like the tensors are some sort of array, unlike in TensorFlow. In TensorFlow, you must run the variable through a session to be able to see the data values.

In comparison, Figure B-2 shows TensorFlow.

In [78]:
```
import tensorflow as tf
import numpy as np

f = tf.constant(np.random.randint(0, 10, 5), shape=(1, 5), dtype=tf.int32)
print(f)
g = tf.constant(np.random.randint(0, 10, 5), shape=(1, 5), dtype=tf.int32)
print(g)
result = A + B
tanh = tf.tanh(tf.to_float(result))


with tf.Session() as sess:
    print("f: {}    g: {}\n".format(sess.run(f), sess.run(g)))
    print("f + g: {}\n".format(sess.run(result)))
    print("tanh(f+g): {}".format(sess.run(tanh)))
```

```
Tensor("Const_37:0", shape=(1, 5), dtype=int32)
Tensor("Const_38:0", shape=(1, 5), dtype=int32)
f: [[5 6 9 7 0]]    g: [[3 9 4 8 9]]

f + g: [[14 16  6  5 16]]

tanh(f+g): [[1.         1.         0.99998784 0.99990916 1.        ]]
```

***Figure B-2.*** *Some tensor operations conducted in TensorFlow. Note that to actually see results, you need to pass everything through a TensorFlow session*

PyTorch has much more functionality in how you can manipulate tensors, so it's worth checking out the documentation if you haven't.

Now, let's move on to creating a PyTorch model in a somewhat advanced, but organized format. Splitting up the definition of the model, the training process, and the testing process into their respective parts will help you understand how these models are created, trained, and evaluated.

You start by applying a convolutional neural network to the MNIST data set in order to showcase the more customizable format of training.

As usual, you begin with your imports (see Figure B-3 and Figure B-4).

```
import torch
import torch.nn as nn
import torchvision
import torchvision.transforms as transforms
import torch.optim as optim
import torch.nn.functional as F
import numpy as np


device = torch.device('cuda:0' if torch.cuda.is_available()
else 'cpu')
```

***Figure B-3.*** *Importing the basic modules needed to create your network*

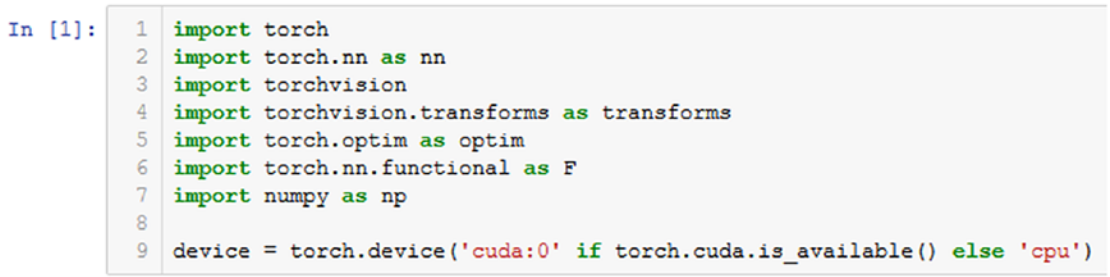



***Figure B-4.*** *The code in Figure B-3 in a Jupyter cell*

In Chapter 3, the code was introduced in a manner similar to basic Keras formatting, so you defined the hyperparameters and loaded your data sets (data loaders in this case) right after importing the modules you need.

Instead, you will now define the model (see Figure B-5 and Figure B-6).

```
class CNN(nn.Module):
    def __init__(self):
        super(CNN, self).__init__()
        self.conv1 = nn.Conv2d(1, 32, 3, 1)
        self.conv2 = nn.Conv2d(32, 64, 3, 1)
        self.dense1 = nn.Linear(12*12*64, 128)
        self.dense2 = nn.Linear(128, num_classes)

    def forward(self, x):
        x = F.relu(self.conv1(x))
        x = F.relu(self.conv2(x))
        x = F.max_pool2d(x, 2, 2)
        x = F.dropout(x, 0.25)
        x = x.view(-1, 12*12*64)
        x = F.relu(self.dense1(x))
        x = F.dropout(x, 0.5)
        x = self.dense2(x)
        return F.log_softmax(x, dim=1)
```

***Figure B-5.*** *Defining the model*

In [2]:

```
class CNN(nn.Module):
    def __init__(self):
        super(CNN, self).__init__()
        self.conv1 = nn.Conv2d(1, 32, 3, 1)
        self.conv2 = nn.Conv2d(32, 64, 3, 1)
        self.dense1 = nn.Linear(12*12*64, 128)
        self.dense2 = nn.Linear(128, num_classes)

    def forward(self, x):
        x = F.relu(self.conv1(x))
        x = F.relu(self.conv2(x))
        x = F.max_pool2d(x, 2, 2)
        x = F.dropout(x, 0.25)
        x = x.view(-1, 12*12*64)
        x = F.relu(self.dense1(x))
        x = F.dropout(x, 0.5)
        x = self.dense2(x)
        return F.log_softmax(x, dim=1)
```

***Figure B-6.*** *The code in Figure B-5 in a Jupyter cell*

With that out of the way, you can define both the training and testing functions (see Figure B-7 and Figure B-8 for the training function, and Figure B-9 and Figure B-10 for the testing function).

```
def train(model, device, train_loader, criterion, optimizer, epoch,
save_dir='model.ckpt'):
    total_step = len(train_loader)
    for i, (images, labels) in enumerate(train_loader):
        images = images.to(device)
        labels = labels.to(device)


        # Forward pass
        outputs = model(images)
        loss = criterion(outputs, labels)


        # Backward and optimize
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()


        if (i+1) % 100 == 0:
            print ('Epoch [{}/{}], Step [{}/{}], Loss:
{:.4f}'.format(epoch+1, num_epochs, i+1, total_step, loss.item()))


    torch.save(model.state_dict(), 'pytorch_mnist_cnn.ckpt')
```

***Figure B-7.*** *The training algorithm. The for loop takes each pair of image and labels and passes them into the GPU as a tensor. They then go into the model, and the gradients are calculated. The information about the epoch and loss are then output*

```
In [6]:
def train(model, device, train_loader, criterion, optimizer, epoch, save_dir='model.ckpt'):
    total_step = len(train_loader)
    for i, (images, labels) in enumerate(train_loader):
        images = images.to(device)
        labels = labels.to(device)

        # Forward pass
        outputs = model(images)
        loss = criterion(outputs, labels)

        # Backward and optimize
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        if (i+1) % 100 == 0:
            print ('Epoch [{}/{}], Step [{}/{}], Loss: {:.4f}'
                   .format(epoch+1, num_epochs, i+1, total_step, loss.item()))

    torch.save(model.state_dict(), 'pytorch_mnist_cnn.ckpt')
```

***Figure B-8.*** *The code in Figure B-7 in a Jupyter cell*

The training function takes in the following parameters:

- **model**: An instance of a model class. In this case, it's an instance of the CNN class defined above.
- **device**: This basically tells PyTorch what device (if the GPU is an option, which GPU to run on, and if not, the CPU is the device) to run on. In this case, you define the device right after the imports.
- **train_loader**: The loader for the training data set. In this case, you use a data_loader because that's how the MNIST data is formatted when importing from torchvision. This data loader contains the training samples for the MNIST data set.
- **criterion**: The loss function to use. Define this before calling the train function.
- **optimizer**: The optimization function to use. Define this before calling the train function.
- **epoch**: What epoch is running. In this case, you call the training function in a for loop while passing in the iteration as the epoch.

The testing function is shown in Figure B-9 and Figure B-10.

```
from sklearn.metrics import roc_auc_score


def test(model, device, test_loader):


    preds = []
    y_true = []
    # Test the model
    model.eval()  # Set model to evaluation mode.
    with torch.no_grad():
        correct = 0
        total = 0
        for images, labels in test_loader:
            images = images.to(device)
            labels = labels.to(device)
            outputs = model(images)
            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()
            detached_pred = predicted.detach().cpu().numpy()
            detached_label = labels.detach().cpu().numpy()
            for f in range(0, len(detached_pred)):
                preds.append(detached_pred[f])
                y_true.append(detached_label[f])


        print('Test Accuracy of the model on the 10000 test images:
{:.2%}'.format(correct / total))


        preds = np.eye(num_classes)[preds]
        y_true = np.eye(num_classes)[y_true]
        auc = roc_auc_score(preds, y_true)
        print("AUC: {:.2%}".format (auc))
```

***Figure B-9.*** *The code for the testing algorithm. Once again, the for loop takes the image and label pairs and passes them through the model to get a prediction. Then, once every pair has a prediction, the AUC score is calculated*

In [7]:
```
from sklearn.metrics import roc_auc_score

def test(model, device, test_loader):

    preds = []
    y_true = []
    # Test the model
    model.eval()  # Set model to evaluation mode.
    with torch.no_grad():
        correct = 0
        total = 0
        for images, labels in test_loader:
            images = images.to(device)
            labels = labels.to(device)
            outputs = model(images)
            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()
            detached_pred = predicted.detach().cpu().numpy()
            detached_label = labels.detach().cpu().numpy()
            for f in range(0, len(detached_pred)):
                preds.append(detached_pred[f])
                y_true.append(detached_label[f])

        print('Test Accuracy of the model on the 10000 test images: {:.2%}'.format(correct / total))

        preds = np.eye(num_classes)[preds]
        y_true = np.eye(num_classes)[y_true]
        auc = roc_auc_score(preds, y_true)
        print("AUC: {:.2%}".format (auc))

```

***Figure B-10.*** *The code in Figure B-9 in a Jupyter cell*

Notice that you use the AUC score as part of the testing metric. You don't have to do this, but it might be a better indicator of the model's performance than plain accuracy, so it was included in this example.

The parameters the model takes in are

- **model**: An instance of a model class. In this case, it's an instance of the CNN class defined above.
- **device**: This basically tells PyTorch what device (if the GPU is an option, which GPU to run on, and if not, the CPU is the device) to run on. In this case, you define the device right after the imports.
- **test_loader**: The loader for the testing data set. In this case, you use a data_loader because that's how the MNIST data is formatted when importing from torchvision. This data loader contains the testing samples for the MNIST data set.

Now you can get to defining your hyperparameters and data loaders, and calling your train and test functions (Figures B-11 through B-13).

```
# Hyperparameters
num_epochs = 15
num_classes = 10
batch_size = 128
learning_rate = 0.001

# Load MNIST data set
train_dataset = torchvision.datasets.MNIST(root='../../data/',
                                           train=True,

transform=transforms.ToTensor(),
                                           download=True)

test_dataset = torchvision.datasets.MNIST(root='../../data/',
                                          train=False,

transform=transforms.ToTensor())

# Data loader
train_loader = torch.utils.data.DataLoader(dataset=train_dataset,
                                           batch_size=batch_size,
                                           shuffle=True)

test_loader = torch.utils.data.DataLoader(dataset=test_dataset,
                                          batch_size=batch_size,
                                          shuffle=False)
```

***Figure B-11.*** *Defining the hyperparameters, loading the MNIST data, and defining the training and testing set data loaders*

```
model = CNN().to(device)
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)

## Training phase

for epoch in range(0, num_epochs):
    train(model, device, train_loader, criterion, optimizer, epoch)

## Testing phase

test(model, device, test_loader)
```

***Figure B-12.*** *Initializing the model and passing it to the GPU, defining your criterion function (cross entropy loss), and defining your optimizer (the Adam optimizer). Then, the training and testing functions are called*

In [8]:

```
#Hyperparameters
num_epochs = 15
num_classes = 10
batch_size = 128
learning_rate = 0.001

#Load MNIST data set
train_dataset = torchvision.datasets.MNIST(root='../../data/',
                                           train=True,
                                           transform=transforms.ToTensor(),
                                           download=True)

test_dataset = torchvision.datasets.MNIST(root='../../data/',
                                          train=False,
                                          transform=transforms.ToTensor())

#Data loader
train_loader = torch.utils.data.DataLoader(dataset=train_dataset,
                                           batch_size=batch_size,
                                           shuffle=True)

test_loader = torch.utils.data.DataLoader(dataset=test_dataset,
                                          batch_size=batch_size,
                                          shuffle=False)



model = CNN().to(device)
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)


## Training phase

for epoch in range(0, num_epochs):
    train(model, device, train_loader, criterion, optimizer, epoch)

## Testing phase


test(model, device, test_loader)
```

***Figure B-13.*** *What the code from Figures B-11 and B-12 should look like after pasting them into a Jupyter cell*

After the training process, you get Figure B-14 and Figure B-15.

```
Epoch [1/15], Step [100/469], Loss: 0.1852
Epoch [1/15], Step [200/469], Loss: 0.0556
Epoch [1/15], Step [300/469], Loss: 0.1524
Epoch [1/15], Step [400/469], Loss: 0.0332
Epoch [2/15], Step [100/469], Loss: 0.0494
Epoch [2/15], Step [200/469], Loss: 0.1003
Epoch [2/15], Step [300/469], Loss: 0.0534
Epoch [2/15], Step [400/469], Loss: 0.0329
Epoch [3/15], Step [100/469], Loss: 0.0387
Epoch [3/15], Step [200/469], Loss: 0.0379
Epoch [3/15], Step [300/469], Loss: 0.0055
Epoch [3/15], Step [400/469], Loss: 0.0183
Epoch [4/15], Step [100/469], Loss: 0.0283
Epoch [4/15], Step [200/469], Loss: 0.0250
Epoch [4/15], Step [300/469], Loss: 0.0210
Epoch [4/15], Step [400/469], Loss: 0.0637
Epoch [5/15], Step [100/469], Loss: 0.0109
Epoch [5/15], Step [200/469], Loss: 0.0091
Epoch [5/15], Step [300/469], Loss: 0.0243
Epoch [5/15], Step [400/469], Loss: 0.0095
Epoch [6/15], Step [100/469], Loss: 0.0310
Epoch [6/15], Step [200/469], Loss: 0.0254
Epoch [6/15], Step [300/469], Loss: 0.0028
```

***Figure B-14.*** *The initial output of the training process*

```
Epoch [12/15], Step [200/469], Loss: 0.0017
Epoch [12/15], Step [300/469], Loss: 0.0006
Epoch [12/15], Step [400/469], Loss: 0.0026
Epoch [13/15], Step [100/469], Loss: 0.0010
Epoch [13/15], Step [200/469], Loss: 0.0002
Epoch [13/15], Step [300/469], Loss: 0.0025
Epoch [13/15], Step [400/469], Loss: 0.0014
Epoch [14/15], Step [100/469], Loss: 0.0000
Epoch [14/15], Step [200/469], Loss: 0.0000
Epoch [14/15], Step [300/469], Loss: 0.0001
Epoch [14/15], Step [400/469], Loss: 0.0064
Epoch [15/15], Step [100/469], Loss: 0.0024
Epoch [15/15], Step [200/469], Loss: 0.0000
Epoch [15/15], Step [300/469], Loss: 0.0017
Epoch [15/15], Step [400/469], Loss: 0.0003
Test Accuracy of the model on the 10000 test images: 98.93%
AUC: 99.40%
```

***Figure B-15.*** *The training process has finished*

Although in your Keras examples you didn't spread apart your training and testing functions (since they're just one line each), more complicated implementations of models involving custom layers, models, and so on can be formatted in a similar fashion to the PyTorch example above.

Hopefully, you understand a bit more on how to implement, train, and test neural networks in PyTorch.

Next, we will explain some of the basic functionality that PyTorch offers in terms of model layers (activations included), loss functions, and optimizers, and then you'll explore PyTorch applications of temporal convolutional neural networks to the data set found in Chapter 7.

## Sequential vs. ModuleList

Similar to Keras, PyTorch has a couple different ways to define the model.

**Sequentially**, as in Figure B-16

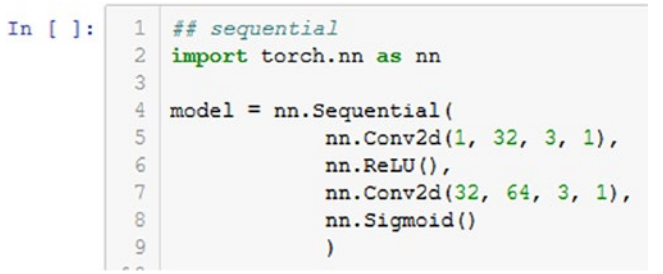



***Figure B-16.*** *A sequential model in PyTorch*

This is similar to the sequential model in Keras, where you add layers one at a time and in order.

**ModuleList**, as in Figure B-17

```
10
11 ## ModuleList
12
13 class ModuleListModel(nn.Module):
14     def __init__(self):
15         super(ModuleListModel, self).__init__()
16         self.conv_1 = nn.Conv2d(1, 32, 3, 1)
17         self.conv_2 = nn.Conv2d(32, 64, 3, 1)
18         self.dense_1 = nn.Linear(64*64, 128)
19         self.output = nn.Linear(128, n_classes)
20
21
22     def forward(self, x):
23         x = nn.functional.relu(self.conv_1(x))
24         x = nn.functional.relu(self.conv_2(x))
25         x = nn.functional.max_pool2d(x, 2, 2)
26         x = nn.functional.dropout(x, 0.25)
27         x = x.view(-1, 64*64)
28         x = nn.functional.reu(self.dense_1(x))
29         x = nn.functional.dropout(x, 0.5)
30         x = nn.functional.log_softmax(self.output(x), dim=1)
31         return x
32
33
34 model = ModuleListModel().to(device)
```

***Figure B-17.*** *A model in PyTorch defined in a ModuleList format*

This is similar to the functional model that you can build in Keras. This is a more customizable way to build your model, and allows you much more flexibility in how you want to build it too.

# Layers

We've covered how to build the models, so let's look at examples of some common layers you can build.

## Conv1d

`torch.nn.Conv1d()`

Check out Chapter 7 for a detailed explanation on how one-dimensional convolutions work.

This layer is a one-dimensional (or temporal) convolutional layer. It basically passes a filter over the one-dimensional input and multiplies the values element-wise to create the output feature map.

These are the parameters that the function takes:

- **in_channels**: The dimensionality of the input space; the number of input nodes.
- **out_channels**: The dimensionality of the output space; the number of output nodes.
- **kernel_size**: The dimensionality of the kernel/filter. An integer **n** makes the dimensions of the kernel **nxn**, and a tuple of two integers allows you to specify the exact dimensions (**height**, **width**).
- **stride**: The number of elements to shift right by after one filter/kernel operation. An integer **n** makes the kernel shift right by that amount. A tuple of two integers allows you to specify (**vertical_shift**, **horizontal_shift).** Default = 1.
- **padding**: The amount of zero padding to add to the layer in the output. An integer **n** pads **n** entries to the rows and columns. A tuple of two integers allows you to specify (**vertical_padding, horizontal_padding**). Default = 0.
- **dilation**: For an explanation on how dilation works, refer to Chapter 7. An integer **n** means a dilation factor of **n**. Default = 1.
- **groups**: Controls the connections between the input and output nodes. Groups=1 means all inputs correlate with all outputs. Groups=2 means there's really two convolutional layers side by side, so half the inputs go to half the outputs. Default = 1.
- **bias**: Whether or not to use bias. Default = True.

## Conv2d

`torch.nn.Conv2d()`

Check out Chapter 3 for a detailed explanation on how the 2D convolutional layer works.

This layer is a two-dimensional convolutional layer. It basically passes a 2D filter over the input and multiplies the values element-wise to create the output feature map.

These are the parameters that the function takes:

- **in_channels**: The dimensionality of the input space; the number of input nodes.
- **out_channels**: The dimensionality of the output space; the number of output nodes.
- **kernel_size**: The dimensionality of the kernel/filter. An integer **n** makes the dimensions of the kernel **nxn**, and a tuple of two integers allows you to specify the exact dimensions (**height**, **width**).
- **stride**: The number of elements to shift right by after one filter/kernel operation. An integer **n** makes the kernel shift right by that amount. A tuple of two integers allows you to specify (**vertical_shift**, **horizontal_shift).** Default = 1.
- **padding**: The amount of zero padding to add to the layer in the output. An integer **n** pads **n** entries to the rows and columns. A tuple of two integers allows you to specify (**vertical_padding, horizontal_padding**). Default = 0.
- **dilation**: For an explanation on how dilation works, refer to Chapter 7. An integer **n** means a dilation factor of **n**. A tuple of two integers allows you to specify (**vertical_dilation**, **horizontal_dilation**). Default = 1.
- **groups**: Controls the connections between the input and output nodes. Groups=1 means all inputs correlate with all outputs. Groups=2 means there's really two convolutional layers side by side, so half the inputs go to half the outputs. Default = 1.
- **bias**: Whether or not to use bias. Default = True.

## Linear

`torch.nn.Linear()`

This is a neural network layer comprised of densely-connected neurons. Basically, every node in this layer is fully connected with the previous and next layers if there are any.

Here are the parameters:

- **in_features**: The size of each input sample; number of inputs.
- **out_features**: The size of each output sample; number of outputs.
- **bias**: Whether or not to use bias. Default = True.

## MaxPooling1D

`torch.nn.MaxPool1d()`

This applies max pooling on a 1D input. To get a better idea of how max pooling works, check out Chapter 3. Max pooling in 1D is similar to max pooling in 2D, except the sliding window only works in one dimension, going from left to right.

This is the list of parameters:

- **kernel_size**: The size of the pooling window. If an integer **n** is given, then the window size of the pooling layer is **1xn.**
- **stride**: Defaults to kernel_size if nothing is passed in. If you pass in an integer, the pooling window moves by integer **n** amount after completing its pooling operation on a set of entries.
- **padding**: An integer **n** representing the zero padding to add on both sides. Default = 0.
- **dilation**: Similar to the dilation factor in the convolutional layer, except with max pooling. Default = 1.
- **return_indices**: If set to True, it will return the indices of the max values along with the outputs. Default = False.
- **ceil_mode**: If set to True, it will use ceil instead of floor to compute the output shape. This comes into play because of the dimensionality reduction involved (a kernel size of **n** will reduce dimensionality by a factor of **n**).

## MaxPooling2D

`torch.nn.MaxPool2d()`

It applies max pooling on a 2D input. To get a better idea of how max pooling works, check out Chapter 3.

This is the list of parameters:

- **kernel_size**: The size of the pooling window. If an integer **n** is given, then the window size of the pooling layer is **1xn.** A tuple of two integers allows you to specify the dimensions as (**height**, **width**).
- **stride**: Defaults to kernel_size if nothing is passed in. If you pass in an integer, the pooling window moves by integer **n** amount after completing its pooling operation on a set of entries. A tuple of two integers allows you to specify (**vertical_shift**, **horizontal_shift)**.
- **padding**: An integer **n** representing the zero padding to add on both sides. A tuple of two integers allows you to specify (**vertical_padding, horizontal_padding**). Default = 0.
- **dilation**: Similar to the dilation factor in the convolutional layer, except with max pooling. An integer **n** means a dilation factor of **n.** A tuple of two integers allows you to specify (**vertical_dilation**, **horizontal_dilation**). Default = 1.
- **return_indices**: If set to True, it will return the indices of the max values along with the outputs. Default = False.
- **ceil_mode**: If set to True, it will use ceil instead of floor to compute the output shape. This comes into play because of the dimensionality reduction involved (a kernel size of **n** will reduce dimensionality by a factor of **n**).

## ZeroPadding2D

`torch.nn.ZeroPad2d()`

Depending on the input, it pads the input sequence with a row and columns of zeroes at the top, left, right, and bottom of the image tensor.

Here is the parameter:

- **padding**: An integer or a tuple of four integers. The integer tells it to add **n** rows of zeroes on the top and bottom of the image tensor, and **n** columns of zeroes. The tuple of four integers is formatted as (padding_left, padding_right, padding_top, padding_bottom), so you can customize even more how you want the layer to add rows or columns of zeroes.

## Dropout

`torch.nn.Dropout()`

What the dropout layer does in PyTorch is take the input and randomly zeroes the elements according to some probability **p** using samples from a Bernoulli distribution. This process is random, so with every forward pass through the model, different elements will be chosen to be zeroed. This process helps with regularization of layer outputs and helps combat overfitting.

Here are the parameters:

- **p**: The probability of an element to be zeroed. Default = 0.5
- **inplace**: If set to True, it will perform the operation in place. Default = False.

You can define this as a layer within the model itself, or apply dropout in the forward function like so:

`torch.nn.functional.Dropout(`**`input, p = 0.5, training=False, inplace=False`**`)`

**Input** is the previous layer, and **training** is a parameter that determines whether or not you want this dropout layer to function outside of training (such as during evaluation).

Figure B-18 shows an example of how you can use this layer in the forward function.

In [ ]:

```
def forward(self, x):
    x = nn.functional.relu(self.conv_1(x))
    x = nn.functional.dropout(x, 0.25)
    x = nn.functional.relu(self.conv_2(x))
    ...
```

***Figure B-18.*** *The dropout layer in the forward function of a model*

So with dropout, you have two ways of applying it, both producing similar outputs. In fact, the layer itself is an extension of the functional version of dropout, which itself is an interface. This is really up to personal preference, since both are still dropout layers and there's no real difference in behavior.

## ReLU

`torch.nn.ReLU()`

ReLU, or "Rectified Linear Unit", performs a simple activation based on the function, as shown in Figure B-19.

$$f(x) = \max(0, x)$$

***Figure B-19.*** *The general formula that ReLU follows*

Here is the parameter:

- **inplace**: If set to True, it will perform the operation in place. Default = False.

For ReLU, the graph can look like Figure B-20.

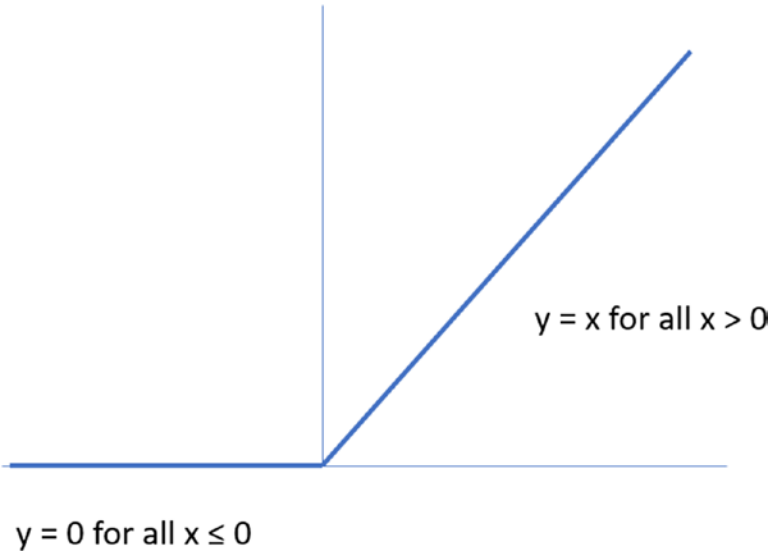



***Figure B-20.*** *The general graph of a ReLU function*

Similarly to dropout, you can define this as a layer within the model itself, or apply ReLU in the forward function like so:

`torch.nn.functional.relu(`**`input, inplace=False`**`)`

**Input** is the previous layer.

Figure B-21 shows an example of how you can use this layer in the `forward` function.

Just like with dropout, you have two ways of applying ReLU, but it all boils down to personal preference.

```
In [ ]:
def forward(self, x):
    x = nn.functional.relu(self.conv_1(x))
    x = nn.functional.dropout(x, 0.25)
    x = nn.functional.relu(self.conv_2(x))
    ...
```

***Figure B-21.*** *The ReLU layer in the forward function of a model*

## Softmax

`torch.nn.Softmax()`

This performs a softmax on the given dimension.

The general formula for softmax is shown in Figure B-22 (K is the number of samples).

$$\sigma(x)_i = \frac{e^{x_i}}{\sum_{j=1}^{K} e^{x_j}} \quad \text{For i = 1,...,K and x = } (x_1, \ldots, x_k) \in R^k$$

***Figure B-22.*** *The general formula for softmax. The parameter i goes up until the total number of samples, which is K*

Here is the parameter:

- **dim**: The dimension to compute softmax along, determined by some integer **n**. This is so every slice along the dimension will sum to 1. Default = None.

You can define this as a layer within the model itself, or apply softmax in the `forward` function like so:

`torch.nn.functional.softmax(`**`input, dim=None, _stacklevel=3`**`)`

**Input** is the previous layer.

Figure B-23 shows an example of how you can use this layer in the `forward` function.

```
In [ ]:
def forward(self, x):
    ...
    x = nn.functional.dropout(x, 0.5)
    x = nn.functional.softmax(self.dense_1(x), dim=1)
    return x
```

***Figure B-23.*** *The softmax layer in the forward function of a model*

However, this doesn't work well if you're using NLLL (negative log likelihood) loss, in which case you should use log_softmax instead.

## Log_Softmax

`torch.nn.LogSoftmax()`

This performs a softmax activation on the given dimension, but passes that through a log function.

The general formula for log_softmax is shown in Figure B-24 (K is the number of samples).

$$\sigma(x)_i = \log\left(\frac{e^{x_i}}{\sum_{j=1}^{K} e^{x_j}}\right) \quad \text{For i = 1,...,K and x = } (x_1, \ldots, x_k) \in R^k$$

***Figure B-24.*** *The general formula for log_softmax. The value i goes up until the total number of samples, K.*

Here is the parameter:

- **dim**: The dimension to compute softmax along, determined by some integer **n**. This is so every slice along the dimension will sum to 1. Default = None.

You can define this as a layer within the model itself, or apply softmax in the `forward` function like so:

`torch.nn.functional.log_softmax(`**`input, dim=None, _stacklevel=3`**`)`

**Input** is the previous layer.

Figure B-25 shows an example of how you can use this layer in the `forward` function.

```
In [ ]:
def forward(self, x):
    ...
    x = nn.functional.dropout(x, 0.5)
    x = nn.functional.log_softmax(self.dense_1(x), dim=1)
    return x
```

***Figure B-25.*** *The log softmax layer in the forward function of a model*

## Sigmoid

`torch.nn.Sigmoid()`

This performs a sigmoid activation.

The sigmoid function does have its uses, primarily because it forces the input to be between 0 and 1, but it is prone to the vanishing gradient problem, and so it is seldom used in hidden layers.

There are no parameters, so it's a simple function to call.

To get an idea of what the equation is like when graphed, refer to Figure B-26.

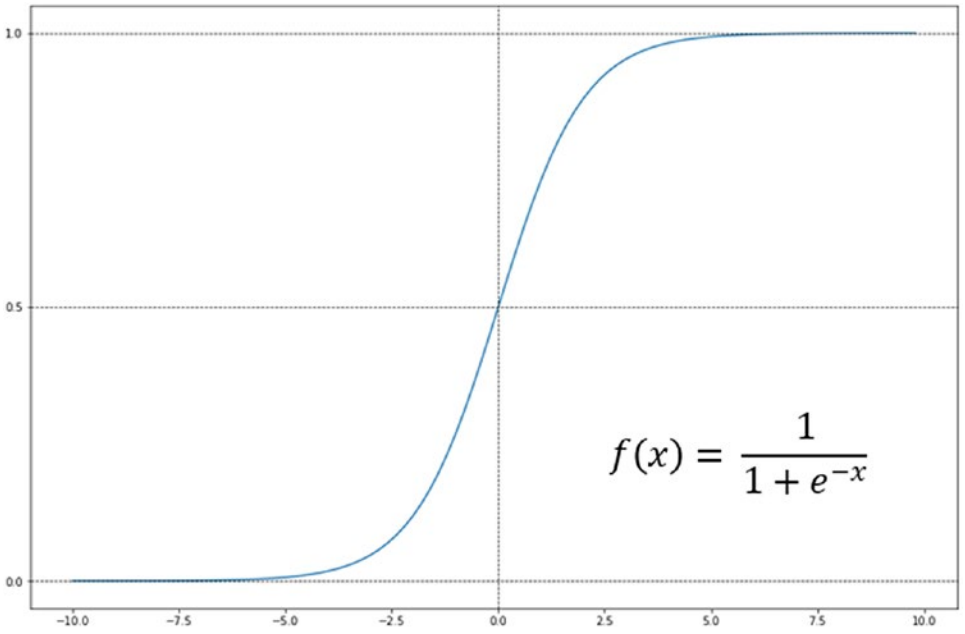



***Figure B-26.*** *The general graph of a sigmoid function*

You can define this as a layer within the model itself, or apply sigmoid in the `forward` function like so:

`torch.nn.functional.sigmoid(`**`input`**`)`

**Input** is the previous layer.

Figure B-27 shows an example of how you can use this layer in the `forward` function.

```
In [ ]:
def forward(self, x):
    ...
    x = nn.functional.dropout(x, 0.5)
    x = nn.functional.sigmoid(self.dense_1(x), dim=1)
    return x
```

***Figure B-27.*** *The sigmoid layer in the forward function of a model*

# Loss Functions

## MSE

`torch.nn.MSELoss()`

If you have questions on the notation for this equation, refer to Chapter 3. The equation is shown in Figure B-28.

$$J(\theta) = \frac{1}{n}\sum_{i=1}^{n}\left(h_\theta\left(x^i\right) - y^i\right)^2$$

***Figure B-28.*** *The general formula for mean squared loss*

Given input $\theta$, the weights, the formula finds the average difference squared between the predicted value and the actual value. The parameter $h_\theta$ represents the model with the weight parameter $\theta$ passed in, so $h_\theta(x^i)$ would give the predicted value for $x^i$ with model's weights $\theta$. The parameter $y^i$ represents the actual prediction for the data point at index i. Lastly, there are n entries in total.

This function has several parameters (two are deprecated):

- **size_average**: (Deprecated in favor of reduction.) The losses are averaged over each loss element in the batch by default (True). If set to False, then the losses are summed for each minibatch instead. Default = True.

- **reduce**: (Deprecated in favor of reduction.) The losses are averaged or summed over observations for each minibatch depending on size_average by default (True). If set to False, then it returns a loss per batch element and ignores size_average. Default = True.
- **reduction**: A string value to specify the type of reduction to be done. Choose between 'none', 'elementwise_mean', or 'sum'. 'none' means no reduction is applied, 'elementwise_mean' will divide the sum of the output by the number of elements in the output, and 'sum' will just sum the output. Default = 'elementwise_mean'. Note: specifying either size_average or reduce will override this parameter.

This loss metric can be used in autoencoders to help evaluate the difference between the reconstructed output and the original. In the case of anomaly detection, this metric can be used to separate the anomalies from the normal data points, since anomalies have a higher reconstruction error.

## Cross Entropy

`torch.nn.CrossEntropyLoss()`

The equation is shown in Figure B-29.

$$J(\theta) = -\frac{1}{n}\sum_{i=0}^{n} y_i * log(h_\theta(x_i)) + (1 - y_i) * \log(1 - h_\theta(x_i))$$

***Figure B-29.*** *The general formula for cross entropy loss*

In this case, **n** is the number of samples in the whole data set. The parameter $h_\theta$ represents the model with the weight parameter $\theta$ passed in, so $h_\theta(x_i)$ would give the predicted value for $x_i$ with model's weights $\theta$. Finally, $y_i$ represents the true labels for data point at index i. The data needs to be regularized to be between 0 and 1, so for categorical cross entropy, it must be piped through a softmax activation layer.
The categorical cross entropy loss is also called **softmax loss**.

Equivalently, you can write the previous equation as Figure B-30.

$$J(\theta) = -\frac{1}{n}\sum_{i=0}^{n}\sum_{j=0}^{m} y_{ij} * \log(h_\theta(x_{ij}))$$

***Figure B-30.*** *An alternate way to write the equation in Figure B-29*

In this case, **m** is the number of classes.

The categorical cross entropy loss is a commonly used metric in classification tasks, especially in computer vision with convolutional neural networks.

This function has several parameters (two are deprecated):

- **weight**: (Optional) A tensor that's the size of the number of classes **n**. This is essentially a weight given to each class so that some classes are weighted more heavily in how they affect the overall loss and optimization problem.
- **size_average**: (Deprecated in favor of reduction.) The losses are averaged over each loss element in the batch by default (True). If set to False, then the losses are summed for each minibatch instead. Default = True.
- **ignore_index**: (Optional) An integer that specifies a target value that is ignored so it does not contribute to the input gradient. If size_average is True, then the loss is averaged over targets that aren't ignored.
- **reduce**: (Deprecated in favor of reduction.) The losses are averaged or summed over observations for each minibatch depending on size_average by default (True). If set to False, it returns a loss per batch element and ignores size_average. Default = True.
- **reduction**: A string value to specify the type of reduction to be done. Choose between 'none', 'elementwise_mean', or 'sum'. 'none' means no reduction is applied, 'elementwise_mean' will divide the sum of the output by the number of elements in the output, and 'sum' will just sum the output. Default = 'elementwise_mean'. Note: Specifying either size_average or reduce will override this parameter.

# Optimizers

## SGD

`torch.optim.SGD()`

This is the **stochastic gradient descent** optimizer, a type of algorithm that aids in the backpropagation process by adjust the weights. It is commonly used as a training algorithm in a variety of machine learning applications, including neural networks.

This function has several parameters:

- **params**: Some iterable of parameters to optimize, or dictionaries with parameter groups. This can be something like `model.parameters()`.
- **lr**: A float value specifying the learning rate.
- **momentum**: (Optional) Some float value specifying the momentum factor. This parameter helps accelerate the optimization steps in the direction of the optimization, and helps reduce oscillations when the local minimum is overshot (refer to Chapter 3 to refresh your understanding on how a loss function is optimized). Default = 0.
- **weight_decay**: A l2_penalty for weights that are too high, helping incentivize smaller model weights. Default = 0.
- **dampening**: The dampening factor for momentum. Default = 0.
- **nesterov**: A Boolean value to determine whether or not to apply Nesterov momentum. Nesterov momentum is a variation of momentum where the gradient is computed not from the current position, but from a position that takes into account the momentum. This is because the gradient always points in the right direction, but the momentum might carry the position too far forward and overshoot. Since this doesn't use the current position but instead some intermediate position that takes into account momentum, the gradient from that position can help correct the current course so that the momentum doesn't carry the new weights too far forward. It essentially helps for more accurate weight updates and helps converge faster. Default = False.

## Adam

`torch.optim.Adam()`

The Adam optimizer is an algorithm that extends upon SGD. It has grown quite popular in deep learning applications in computer vision and in natural language processing.

This function has several parameters:

- **params**: Some iterable of parameters to optimize, or dictionaries with parameter groups. This can be something like `model.parameters()`.
- **lr**: A float value specifying the learning rate. Default = 0.001 (or 1e-3).
- **betas**: (Optional) A tuple of two floats to define the beta values beta_1 and beta_2. The paper describes good results with (0.9, 0.999) respectively, which is also the default value.
- **eps**: (Optional). Some float value where epsilon **e** >= 0. Epsilon is some small number, described as 10E-8 in the paper, to help prevent division by 0. Default is 1e-8.
- **weight_decay**: A l2_penalty for weights that are too high, helping incentivize smaller model weights. Default = 0.
- **amsgrad**: A Boolean on whether or not to apply the AMSGrad version of this algorithm. For more details on the implementation of this algorithm, check out "On the Convergence of Adam and Beyond." Default=False.

## RMSProp

`torch.optim.RMSprop()`

RMSprop is a good algorithm for recurrent neural networks. RMSprop is a gradient-based optimization technique developed to help address the problem of gradients becoming too large or too small. RMSprop helps combat this problem by normalizing the gradient itself using the average of the squared gradients. In Chapter 7, it's explained that one of the problems with RNNs is the vanishing/exploding gradient problem,

leading to the development of LSTMs and GRU networks. And so it's of no surprise that RMSprop pairs well with recurrent neural networks.

This function has several parameters:

- **params**: Some iterable of parameters to optimize, or dictionaries with parameter groups. This can be something like `model.parameters()`.
- **lr**: A float value specifying the learning rate. Default = 0.01 (or 1e-2).
- **momentum**: (Optional). Some float value specifying the momentum factor. This parameter helps accelerate the optimization steps in the direction of the optimization, and helps reduce oscillations when the local minimum is overshot (refer to Chapter 3 to refresh your understanding on how a loss function is optimized). Default = 0.
- **alpha**: (Optional) A smoothing constant. Default = 0.99
- **eps**: (Optional). Some float value where epsilon **e** >= 0. Epsilon is some small number, described as 10E-8 in the paper, to help prevent division by 0. Default is 1e-8.
- **centered**: (Optional) If True, then compute the centered RMSprop and have the gradient normalized by an estimation of its variance. Default = False.
- **weight_decay**: An l2_penalty for weights that are too high, helping incentivize smaller model weights. Default = 0.

Hopefully by now you understand how PyTorch works by looking at some of the functionality that it offers. You built and applied a model to the MNIST data set in an organized format, and you looked at some of the basics of PyTorch by learning about the layers, how models are constructed, how activations are performed, and what the loss functions and optimizers are.

# Temporal Convolutional Network in PyTorch

Now, you will look at an example of using PyTorch to construct a temporal convolutional network and apply it to the credit card data set from Chapter 7.

## Dilated Temporal Convolutional Network

The particular TCN you will reconstruct in PyTorch is the dilated TCN in Chapter 7.

Once again, you begin with your imports and define your device (Figure B-31 and Figure B-32).

```
import numpy as np
import pandas as pd
import keras
from sklearn.model_selection import train_test_split
from sklearn.preprocessing.data import StandardScaler
import torch
import torch.nn as nn
import torch.nn.functional as F



# Hyperparameters
num_epochs = 30
num_classes = 2
learning_rate = 0.002


device = torch.device('cuda:0' if torch.cuda.is_available() else
'cpu')
```

***Figure B-31.*** *Importing the necessary modules*

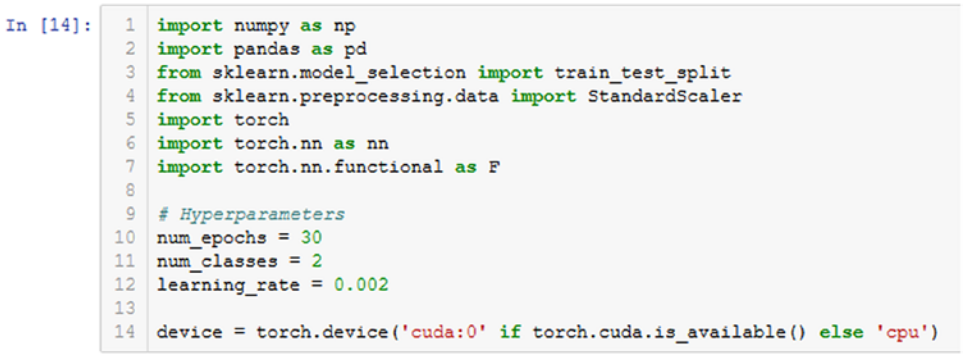



***Figure B-32.*** *The code in Figure B-31 in a Jupyter cell*

Next, you load your data set (Figure B-33).

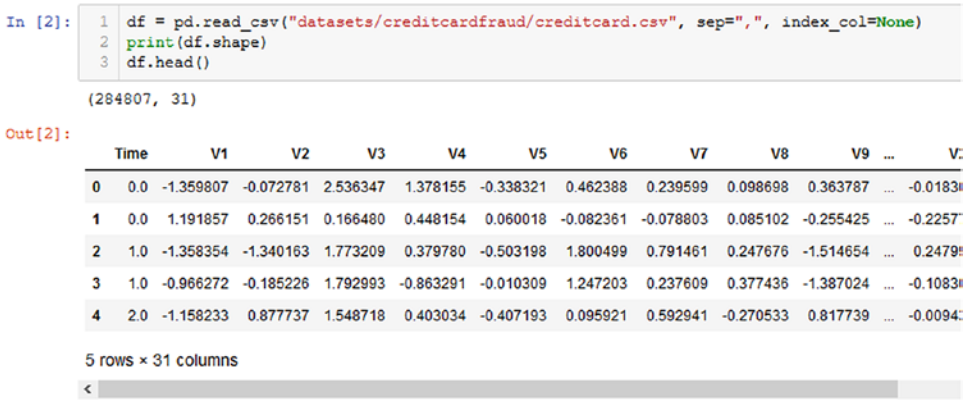
```
df = pd.read_csv("datasets/creditcardfraud/creditcard.csv",
sep=",", index_col=None)

print(df.shape)

df.head()
```

***Figure B-33.*** *Loading your data set and displaying the first five rows*

The output should look somewhat like Figure B-34.



```
In [2]:
1 df = pd.read_csv("datasets/creditcardfraud/creditcard.csv", sep=",", index_col=None)
2 print(df.shape)
3 df.head()

(284807, 31)

Out[2]:
```

| | Time | V1 | V2 | V3 | V4 | V5 | V6 | V7 | V8 | V9 | ... | V: |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0.0 | -1.359807 | -0.072781 | 2.536347 | 1.378155 | -0.338321 | 0.462388 | 0.239599 | 0.098698 | 0.363787 | ... | -0.0183 |
| 1 | 0.0 | 1.191857 | 0.266151 | 0.166480 | 0.448154 | 0.060018 | -0.082361 | -0.078803 | 0.085102 | -0.255425 | ... | -0.2257 |
| 2 | 1.0 | -1.358354 | -1.340163 | 1.773209 | 0.379780 | -0.503198 | 1.800499 | 0.791461 | 0.247676 | -1.514654 | ... | 0.2479 |
| 3 | 1.0 | -0.966272 | -0.185226 | 1.792993 | -0.863291 | -0.010309 | 1.247203 | 0.237609 | 0.377436 | -1.387024 | ... | -0.1083 |
| 4 | 2.0 | -1.158233 | 0.877737 | 1.548718 | 0.403034 | -0.407193 | 0.095921 | 0.592941 | -0.270533 | 0.817739 | ... | -0.0943 |

5 rows × 31 columns

***Figure B-34.*** *The output of the code in Figure B-33*

You need to standardize the values for Time and for Amount since they can get large. Everything else has already been standardized in the data set. Run the code in Figure B-35.

```
df['Amount'] =
StandardScaler().fit_transform(df['Amount'].values.reshape(-1, 1))

df['Time'] =
StandardScaler().fit_transform(df['Time'].values.reshape(-1, 1))

df.tail()
```

***Figure B-35.*** *Standardizing the values in the columns Amount and Time*

The output should look somewhat like Figure B-36.

```
In [3]:
df['Amount'] = StandardScaler().fit_transform(df['Amount'].values.reshape(-1, 1))
df['Time'] = StandardScaler().fit_transform(df['Time'].values.reshape(-1, 1))
df.tail()
```

Out[3]:

| | Time | V1 | V2 | V3 | V4 | V5 | V6 | V7 | V8 | |
|---|---|---|---|---|---|---|---|---|---|---|
| 284802 | 1.641931 | -11.881118 | 10.071785 | -9.834783 | -2.066656 | -5.364473 | -2.606837 | -4.918215 | 7.305334 | 1.91 |
| 284803 | 1.641952 | -0.732789 | -0.055080 | 2.035030 | -0.738589 | 0.868229 | 1.058415 | 0.024330 | 0.294869 | 0.58 |
| 284804 | 1.641974 | 1.919565 | -0.301254 | -3.249640 | -0.557828 | 2.630515 | 3.031260 | -0.296827 | 0.708417 | 0.43 |
| 284805 | 1.641974 | -0.240440 | 0.530483 | 0.702510 | 0.689799 | -0.377961 | 0.623708 | -0.686180 | 0.679145 | 0.39 |
| 284806 | 1.642058 | -0.533413 | -0.189733 | 0.703337 | -0.506271 | -0.012546 | -0.649617 | 1.577006 | -0.414650 | 0.48 |

5 rows × 31 columns

***Figure B-36.*** *The output of the code in figure B-35.*

Now you define your normal and anomaly data sets (see Figure B-37.

```
anomalies = df[df["Class"] == 1]

normal = df[df["Class"] == 0]


anomalies.shape, normal.shape
```

***Figure B-37.*** *Defining the anomaly and normal data sets*

The output should look like Figure B-38.

```
In [4]:
anomalies = df[df["Class"] == 1]
normal = df[df["Class"] == 0]

anomalies.shape, normal.shape

Out[4]: ((492, 31), (284315, 31))
```

***Figure B-38.*** *The output of the code in Figure B-37*

After isolating the anomalies from the normal data, let's create your training and testing sets (see Figure B-39.

```
for f in range(0, 20):
    normal = normal.iloc[np.random.permutation(len(normal))]



data_set = pd.concat([normal[:10000], anomalies])


x_train, x_test = train_test_split(data_set, test_size = 0.4,
random_state = 42)


x_train = x_train.sort_values(by=['Time'])
x_test = x_test.sort_values(by=['Time'])


y_train = x_train["Class"]
y_test = x_test["Class"]


x_train.head(10)
```

***Figure B-39.*** *The creation of the training and testing data sets*

The output should look somewhat like Figure B-40.

In [5]:

```
for f in range(0, 20):
    normal = normal.iloc[np.random.permutation(len(normal))]


data_set = pd.concat([normal[:10000], anomalies])

x_train, x_test = train_test_split(data_set, test_size = 0.4, random_state = 42)

x_train = x_train.sort_values(by=['Time'])
x_test = x_test.sort_values(by=['Time'])

y_train = x_train["Class"]
y_test = x_test["Class"]

x_train.head(10)
```

Out[5]:

| | Time | V1 | V2 | V3 | V4 | V5 | V6 | V7 | V8 | |
|---|---|---|---|---|---|---|---|---|---|---|
| **8** | -1.996436 | -0.894286 | 0.286157 | -0.113192 | -0.271526 | 2.669599 | 3.721818 | 0.370145 | 0.851084 | -0.392 |
| **99** | -1.995151 | 1.232996 | 0.189454 | 0.491040 | 0.633673 | -0.511574 | -0.990609 | 0.066240 | -0.196940 | 0.075 |
| **118** | -1.994983 | -0.997176 | 0.228365 | 1.715340 | -0.420067 | 0.560838 | 0.564725 | 0.846047 | 0.197491 | -0.097 |
| **177** | -1.994182 | 1.194066 | -0.072582 | 0.635286 | 0.768616 | -0.735584 | -0.673466 | -0.146299 | -0.065653 | 0.646 |
| **225** | -1.993488 | -2.687978 | 4.390230 | -2.360483 | 0.360829 | 1.310192 | -1.645253 | 2.327776 | -1.727825 | 4.324 |
| **259** | -1.992729 | 0.726749 | -0.528042 | 0.050366 | 1.373621 | -0.124122 | 0.415688 | 0.259555 | 0.085114 | -0.003 |
| **356** | -1.991087 | 1.260328 | 0.299161 | 0.527681 | 0.614899 | -0.420592 | -0.977533 | 0.108485 | -0.244502 | -0.058 |
| **374** | -1.990834 | 1.124355 | -0.132953 | 0.588467 | 0.804871 | -0.726266 | -0.521875 | -0.167010 | 0.059298 | 0.368 |
| **379** | -1.990729 | -1.092301 | 0.430750 | 1.249785 | 0.429757 | 1.272076 | 0.548203 | -0.120592 | 0.452571 | -0.414 |
| **400** | -1.990476 | -0.695818 | 0.581773 | 2.378180 | 0.063396 | 0.329119 | -0.449865 | 1.269104 | -0.758363 | 0.381 |

10 rows × 31 columns

***Figure B-40.*** *The output of the code in Figure B-39*

After defining your data sets, you need to reshape the values so that your neural network can accept them (see Figure B-41).

```
x_train = np.array(x_train).reshape(x_train.shape[0], 1,
x_train.shape[1])

x_test = np.array(x_test).reshape(x_test.shape[0], 1,
x_test.shape[1])


y_train = np.array(y_train).reshape(y_train.shape[0] , 1)

y_test = np.array(y_test).reshape(y_test.shape[0], 1)


print("Shapes:\nx_train:%s\ny_train:%s\n" % (x_train.shape,
y_train.shape))

print("x_test:%s\ny_test:%s\n" % (x_test.shape,
y_test.shape))
```

***Figure B-41.*** *Reshaping the training and testing data sets so you can pass them into the model*

The output should look like Figure B-42.

```
In [6]:
x_train = np.array(x_train).reshape(x_train.shape[0], 1, x_train.shape[1])
x_test = np.array(x_test).reshape(x_test.shape[0], 1, x_test.shape[1])

y_train = np.array(y_train).reshape(y_train.shape[0] , 1)
y_test = np.array(y_test).reshape(y_test.shape[0], 1)

print("Shapes:\nx_train:%s\ny_train:%s\n" % (x_train.shape, y_train.shape))
print("x_test:%s\ny_test:%s\n" % (x_test.shape, y_test.shape))

Shapes:
x_train:(6295, 1, 31)
y_train:(6295, 1)

x_test:(4197, 1, 31)
y_test:(4197, 1)
```

***Figure B-42.*** *The output of the code in Figure B-41*

Now you can define your model (Figure B-43 and Figure B-44).

```
class TCN(nn.Module):

    def __init__(self):

        super(TCN, self).__init__()


        self.conv_1 = nn.Conv1d(1, 128, kernel_size=2, dilation=1,
padding=((2-1) * 1))

        self.conv_2 = nn.Conv1d(128, 128, kernel_size=2, dilation=2,
padding=((2-1) * 2))

        self.conv_3 = nn.Conv1d(128, 128, kernel_size=2, dilation=4,
padding=((2-1) * 4))

        self.conv_4 = nn.Conv1d(128, 128, kernel_size=2, dilation=8,
padding=((2-1) * 8))

        self.dense_1 = nn.Linear(31*128  , 128)

        self.dense_2 = nn.Linear(128, num_classes)
```

***Figure B-43.*** *The first part of the TCN class*

```
def forward(self, x):
        x = self.conv_1(x)
        x = x[:, :, :-self.conv_1.padding[0]]
        x = F.relu(x)
        x = F.dropout(x, 0.05)
        x = self.conv_2(x)
        x = x[:, :, :-self.conv_2.padding[0]]
        x = F.relu(x)
        x = F.dropout(x, 0.05)
        x = self.conv_3(x)
        x = x[:, :, :-self.conv_3.padding[0]]
        x = F.relu(x)
        x = F.dropout(x, 0.05)
        x = self.conv_4(x)
        x = x[:, :, :-self.conv_4.padding[0]]
        x = F.relu(x)
        x = F.dropout(x, 0.05)
        x = x.view(-1, 31*128)
        x = F.relu(self.dense_1(x))
        x = self.dense_2(x)
        return F.log_softmax(x, dim=1)
```

***Figure B-44.*** *The forward function in the TCN class*

The code for the model should look like Figure B-45.

In [13]:

```
class TCN(nn.Module):
    def __init__(self):
        super(TCN, self).__init__()

        self.conv_1 = nn.Conv1d(1, 128, kernel_size=2, dilation=1, padding=((2-1) * 1))
        self.conv_2 = nn.Conv1d(128, 128, kernel_size=2, dilation=2, padding=((2-1) * 2))
        self.conv_3 = nn.Conv1d(128, 128, kernel_size=2, dilation=4, padding=((2-1) * 4))
        self.conv_4 = nn.Conv1d(128, 128, kernel_size=2, dilation=8, padding=((2-1) * 8))
        self.dense_1 = nn.Linear(31*128  , 128)
        self.dense_2 = nn.Linear(128, num_classes)

    def forward(self, x):
        x = self.conv_1(x)
        x = x[:, :, :-self.conv_1.padding[0]]
        x = F.relu(x)
        x = F.dropout(x, 0.05)
        x = self.conv_2(x)
        x = x[:, :, :-self.conv_2.padding[0]]
        x = F.relu(x)
        x = F.dropout(x, 0.05)
        x = self.conv_3(x)
        x = x[:, :, :-self.conv_3.padding[0]]
        x = F.relu(x)
        x = F.dropout(x, 0.05)
        x = self.conv_4(x)
        x = x[:, :, :-self.conv_4.padding[0]]
        x = F.relu(x)
        x = F.dropout(x, 0.05)
        x = x.view(-1, 31*128)
        x = F.relu(self.dense_1(x))
        x = self.dense_2(x)
        return F.log_softmax(x, dim=1)
```

***Figure B-45.*** *The code from Figures B-43 and B-44 in a Jupyter cell. This defines the entire model*

Now you can define your training and testing functions (Figure B-46, Figure B-48, and Figure B-49).

```
def train(model, device, x_train, y_train, criterion, optimizer,
epoch, save_dir='TCN_CreditCard_PyTorch.ckpt'):
    total_step = len(x_train)

    x_train = torch.Tensor(x_train).cuda().float()
    y_train = torch.Tensor(y_train).cuda().long()

    x_train.to(device)
    y_train.to(device)

    # Forward pass
    outputs = model(x_train)
    loss = criterion(outputs, y_train.squeeze(1))

    # Backward and optimize
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()

    print('Epoch {}/{}, Loss: {:.4f}'.format(epoch+1,
num_epochs, loss.item()))

    torch.save(model.state_dict(), save_dir)
```

***Figure B-46.*** *The training function. Since you don't have data loaders, you pass in x_train and y_train directly into the GPU after converting them to tensors. The inputs then pass through, and the gradients are calculated.*

In [12]:

```
def train(model, device, x_train, y_train, criterion, optimizer, epoch, save_dir='TCN_CreditCard_PyTorch.ckpt'):
    total_step = len(x_train)

    x_train = torch.Tensor(x_train).cuda().float()
    y_train = torch.Tensor(y_train).cuda().long()

    x_train.to(device)
    y_train.to(device)

    # Forward pass
    outputs = model(x_train)
    loss = criterion(outputs, y_train.squeeze(1))

    # Backward and optimize
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()

    print('Epoch {}/{}, Loss: {:.4f}'.format(epoch+1, num_epochs, loss.item()))

    torch.save(model.state_dict(), save_dir)
```

***Figure B-47.*** *The code from Figure B-46 in a Jupyter cell*

```
from sklearn.metrics import roc_auc_score

def test(model, device, x_test, y_test):
    preds = []
    y_true = []

    # Set model to evaluation mode.
    model.eval()
    with torch.no_grad():
        correct = 0
        total = 0

        x_test = torch.Tensor(x_test).cuda().float()
        y_test = torch.Tensor(y_test).cuda().long()

        x_test = x_test.to(device)
        y_test = y_test.to(device)
        y_test = y_test.squeeze(1)
        outputs = model(x_test)
        _, predicted = torch.max(outputs.data, 1)
        total += y_test.size(0)
        correct += (predicted == y_test).sum().item()
        detached_pred = predicted.detach().cpu().numpy()
        detached_label = y_test.detach().cpu().numpy()
```

***Figure B-48.*** *The testing function. Since there are no data loaders, the testing sets must be converted into a tensor and passed into a GPU before being able to make predictions on them. The AUC score is then generated along with an accuracy value*

The rest of the testing function code is shown in Figure B-49.

```
        for f in range(0, len(detached_label)):
                    preds.append(detached_pred[f])
                    y_true.append(detached_label[f])


                print('Test Accuracy of the model on the 10000
        test images: {:.2%}'.format(correct / total))


                preds = np.eye(num_classes)[preds]
                y_true = np.eye(num_classes)[y_true]
                auc = roc_auc_score(np.round(preds), y_true)
                print("AUC: {:.2%}".format (auc))
```

***Figure B-49.*** *The rest of the testing function. This deals with calculating the AUC score and accuracy value*

The entire test function should look like Figure B-50.

In [218]:

```
from sklearn.metrics import roc_auc_score

def test(model, device, x_test, y_test):
    preds = []
    y_true = []

    # Set model to evaluation mode.
    model.eval()
    with torch.no_grad():
        correct = 0
        total = 0

        x_test = torch.Tensor(x_test).cuda().float()
        y_test = torch.Tensor(y_test).cuda().long()

        x_test = x_test.to(device)
        y_test = y_test.to(device)
        y_test = y_test.squeeze(1)
        outputs = model(x_test)
        _, predicted = torch.max(outputs.data, 1)
        total += y_test.size(0)
        correct += (predicted == y_test).sum().item()
        detached_pred = predicted.detach().cpu().numpy()
        detached_label = y_test.detach().cpu().numpy()
        for f in range(0, len(detached_label)):
            preds.append(detached_pred[f])
            y_true.append(detached_label[f])

        print('Test Accuracy of the model on the 10000 test images: {:.2%}'.format(correct / total))

        preds = np.eye(num_classes)[preds]
        y_true = np.eye(num_classes)[y_true]
        auc = roc_auc_score(np.round(preds), y_true)
        print("AUC: {:.2%}".format (auc))
```

***Figure B-50.*** *The entire test function, comprised of code from Figures B-48 and B-49*

Finally, you can train our model as shown in Figure B-51.

```
model = TCN().to(device)

criterion = nn.CrossEntropyLoss()

optimizer = torch.optim.Adam(model.parameters(),
lr=learning_rate)



## Training phase


for epoch in range(0, num_epochs):

    train(model, device, x_train, y_train, criterion,
optimizer, epoch)
```

***Figure B-51.*** *Initializing the TCN model, defining the criterion as the cross entropy loss, and defining the optimizer (Adam optimizer)*

The output should look somewhat like Figure B-52.

```
In [15]:
model = TCN().to(device)
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)


## Training phase

for epoch in range(0, num_epochs):
    train(model, device, x_train, y_train, criterion, optimizer, epoch)

Epoch 1/30, Loss: 0.7319
Epoch 2/30, Loss: 0.5128
Epoch 3/30, Loss: 0.3454
Epoch 4/30, Loss: 0.2430
Epoch 5/30, Loss: 0.1245
Epoch 6/30, Loss: 0.1041
Epoch 7/30, Loss: 0.0684
Epoch 8/30, Loss: 0.0737
Epoch 9/30, Loss: 0.0646
Epoch 10/30, Loss: 0.0672
Epoch 11/30, Loss: 0.0650
Epoch 12/30, Loss: 0.0545
Epoch 13/30, Loss: 0.0526
Epoch 14/30, Loss: 0.0428
Epoch 15/30, Loss: 0.0403
Epoch 16/30, Loss: 0.0409
Epoch 17/30, Loss: 0.0453
Epoch 18/30, Loss: 0.0366
Epoch 19/30, Loss: 0.0357
Epoch 20/30, Loss: 0.0358
Epoch 21/30, Loss: 0.0351
Epoch 22/30, Loss: 0.0347
Epoch 23/30, Loss: 0.0342
Epoch 24/30, Loss: 0.0333
Epoch 25/30, Loss: 0.0329
Epoch 26/30, Loss: 0.0322
Epoch 27/30, Loss: 0.0316
Epoch 28/30, Loss: 0.0314
Epoch 29/30, Loss: 0.0307
Epoch 30/30, Loss: 0.0304
```

***Figure B-52.*** *The output of the training process*

And now you can evaluate your model (see Figure B-53).

```
## Testing phase


test(model, device, x_test,
y_test)
```

***Figure B-53.*** *Calling the test function*

The output should look somewhat like Figure B-54.

```
In [16]:
1  ## Testing phase
2
3  test(model, device, x_test, y_test)

Test Accuracy of the model on the 10000 test images: 99.14%
AUC: 98.98%
```

***Figure B-54.*** *The output AUC value of the testing function*

With the end of this example, you will have created a TCN in both Keras and PyTorch. This way, you'll have a good way to compare how the model is built, trained, and evaluated in both frameworks, allowing you to observe the similarities and differences in how both frameworks handle those processes.

By now, you should have a better understanding of how PyTorch works, especially with how it's meant to be more intuitive. Think back to the training function and the process of converting the data sets, passing them through the GPU and through the model, calculated the gradients, and backpropagating. Though it's not abstracted away from you like in Keras, it still makes logical sense in that the functions called directly correlate to the training process of a neural network.

# Summary

PyTorch is a low-level tool that allows you to quickly create, train, and test your own deep learning models, although it is more complicated than doing the same in Keras. However, it offers you much more functionality, flexibility, and customizability compared to Keras, and compared to TensorFlow, it is much lighter on syntax. With PyTorch, you don't have to worry about switching frameworks as you get more advanced because of the functionality that it offers, making it a great tool to use when conduct deep learning research. PyTorch should be enough for most of your needs as you become more experienced with deep learning, and using either PyTorch or TensorFlow (or tf.keras + TensorFlow) is just a matter of personal preference.

# Index

## A



## B






S. Alla and S. K. Adari, *Beginning Anomaly Detection Using Python-Based Deep Learning*,
https://doi.org/10.1007/978-1-4842-5177-5

## C



## D

## E



## F



## G



## H



## I, J

## K

## L



## M



## N

## O



## P, Q



## R

## S



## T



## U

## V, W, X, Y, Z