Part 2 - Building and Training Models

In this notebook we will cover the following topics

- Creating models from scratch with Keras
- Training the model
- Looking at the results of the training

In [1]:
```
import numpy as np
np.warnings.filterwarnings('ignore')  # Hide np.floating warning
import holoviews as hv
hv.extension('bokeh')
```

In [9]:
```
# Prevent TensorFlow from grabbing all the GPU memory
import tensorflow as tf
config = tf.ConfigProto()
config.gpu_options.allow_growth=True
sess = tf.Session(config=config)
```

```
gpu_options {
  allow_growth: true
}
```

Loading Data

Using the pattern we saw in the last notebook, we can load and transform the CIFAR10 data for deep learning.

In [4]:
```
from keras.datasets import cifar10
import keras.utils

(x_train, y_train), (x_test, y_test) = cifar10.load_data()

# Save an unmodified copy of y_test for later, flattened to one column
y_test_true = y_test[:,0].copy()

x_train = x_train.astype('float32')
x_test = x_test.astype('float32')
x_train /= 255
x_test /= 255

num_classes = 10
y_train = keras.utils.to_categorical(y_train, num_classes)
y_test = keras.utils.to_categorical(y_test, num_classes)

# The data only has numeric categories so we also have the string labels bel
ow
cifar10_labels = np.array(['airplane', 'automobile', 'bird', 'cat', 'deer',
                           'dog', 'frog', 'horse', 'ship', 'truck'])
```

```
Using TensorFlow backend.
```

Creating a Model

The simplest way to define a deep learning model in Keras is using the Sequential class (https://keras.io/getting-started/sequential-model-guide/), which holds a stack of layers that are executed in sequence.

Keras has an extensive catalog of layers (https://keras.io/layers/about-keras-layers/), making it very easy to recreate almost any network you find in the literature. The VGG16-like networks we will use in this tutorial have the following kinds of layers:

- Conv2D - 2D convolutions, useful for image networks
- MaxPooling2D - Pooling of adjacent values using the `max()` function in 2 dimensions, also useful in image networks
- Flatten - Turn any shape input in to a flat, 1D output. Often used to transition to dense layers
- Dense - The traditional neural network layer, where each output is a weighted sum of input layers + offset with an activation function.

Keras also has a large list of supported activation functions (https://keras.io/activations/). For all of these examples, we will use the `relu` function as it has good performance.

We begin by importing the necessary classes:

In [5]:
```
from keras import Sequential
from keras.layers import Conv2D, MaxPooling2D, Flatten, Dense
```

Creating a Keras model has the following structure:

- Create empty model
- Add layers in order, setting the input_shape for the first layer
- Finish by compiling the model with a loss function, an optimizer, and a list of metrics to compute during fitting

The choice of loss function (https://keras.io/losses/) depends on the kind of model we are training. Since we are doing categorization with more than two categories, `categorical_crossentropy` is preferred.

The choice of optimizer (https://keras.io/optimizers/) is less straightforward. We're using `Adadelta` because it is self-tuning and works pretty well on this problem.

Metrics are functions that score your model, but are not used to optimize it. The most common metric is accuracy, so we include it here.

In [10]:
```
model = Sequential()

### Convolution and max pool layers

# Group 1: Convolution
model.add(Conv2D(32, kernel_size=(3, 3),
                 activation='relu',
                 input_shape=x_train.shape[1:]))
model.add(Conv2D(32, (3, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))

# Group 2: Convolution
model.add(Conv2D(64, kernel_size=(3, 3),
                 activation='relu',
                 input_shape=x_train.shape[1:]))
model.add(Conv2D(64, (3, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))

# Group 3: Dense layers
model.add(Flatten())
model.add(Dense(128, activation='relu'))
model.add(Dense(num_classes, activation='softmax'))

model.compile(loss=keras.losses.categorical_crossentropy,
              optimizer=keras.optimizers.Adadelta(),
              metrics=['accuracy'])
```

We can inspect various properties of the model, such as the number of free parameters:

In [11]:
```
model.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
conv2d_5 (Conv2D)            (None, 30, 30, 32)        896       
_________________________________________________________________
conv2d_6 (Conv2D)            (None, 28, 28, 32)        9248      
_________________________________________________________________
max_pooling2d_3 (MaxPooling2 (None, 14, 14, 32)        0         
_________________________________________________________________
conv2d_7 (Conv2D)            (None, 12, 12, 64)        18496     
_________________________________________________________________
conv2d_8 (Conv2D)            (None, 10, 10, 64)        36928     
_________________________________________________________________
max_pooling2d_4 (MaxPooling2 (None, 5, 5, 64)          0         
_________________________________________________________________
flatten_2 (Flatten)          (None, 1600)              0         
_________________________________________________________________
dense_3 (Dense)              (None, 128)               204928    
_________________________________________________________________
dense_4 (Dense)              (None, 10)                1290      
=================================================================
Total params: 271,786
Trainable params: 271,786
Non-trainable params: 0
_________________________________________________________________
```

Here we see that the majority of free parameters are introduced at the point where we switch from the convolutional layers to the dense layers. If we want to reduce the size of this model, we will either need to reduce the size of the dense layer or reduce the number of convolution kernels.

Training a Model

To train a model, we use the `fit()` method on a compiled model:

In [8]:
```
%%time
history = model.fit(x_train, y_train,
          batch_size=256,
          epochs=5,
          verbose=1,
          validation_data=(x_test, y_test))
```

```
Train on 50000 samples, validate on 10000 samples
Epoch 1/5
50000/50000 [==============================] - 11s 221us/step - loss: 2.0699
- acc: 0.2492 - val_loss: 1.7479 - val_acc: 0.3842
Epoch 2/5
50000/50000 [==============================] - 9s 183us/step - loss: 1.6068 -
acc: 0.4272 - val_loss: 1.5601 - val_acc: 0.4268
Epoch 3/5
50000/50000 [==============================] - 9s 183us/step - loss: 1.3832 -
acc: 0.5078 - val_loss: 1.4178 - val_acc: 0.4865
Epoch 4/5
50000/50000 [==============================] - 9s 183us/step - loss: 1.2233 -
acc: 0.5671 - val_loss: 1.2408 - val_acc: 0.5531
Epoch 5/5
50000/50000 [==============================] - 9s 183us/step - loss: 1.1020 -
acc: 0.6138 - val_loss: 1.1234 - val_acc: 0.5997
CPU times: user 34.6 s, sys: 7.39 s, total: 42 s
Wall time: 48.1 s
```

The `epochs` value controls how many passes through the training data are taken. The `batch_size` determines how many training examples are processed in parallel. The model parameters are updated between each batch using backpropagation according to the optimizer's strategy. Batch size affects both training performance and model quality, as we'll discuss later.

The validation data is not used by the optimizer for training, but it is scored between each epoch to give an independent assessment of the model quality. The results for the validation data are what you should keep an eye on to understand how well the model is generalizing. In the next notebook, we'll look more closely at how to interpret differences in accuracy between training and validation data.

Note that the model object retains its state after training. If we wanted additional rounds of training, we could call `fit()` again, and it would pick up where the last fit left off:

In [12]:
```
model.fit(x_train, y_train,
          batch_size=256,
          epochs=2,
          verbose=1,
          validation_data=(x_test, y_test))
```

```
Train on 50000 samples, validate on 10000 samples
Epoch 1/2
50000/50000 [==============================] - 9s 189us/step - loss: 2.0381 -
acc: 0.2629 - val_loss: 1.9802 - val_acc: 0.3253
Epoch 2/2
50000/50000 [==============================] - 9s 183us/step - loss: 1.5815 -
acc: 0.4337 - val_loss: 1.5312 - val_acc: 0.4355
```

Out[12]: <keras.callbacks.History at 0x7f0599c88630>

In [13]:
```
model.fit(x_train, y_train,
          batch_size=256,
          epochs=5,          # changing the epochs to see different values
          verbose=1,
          validation_data=(x_test, y_test))
```

```
Train on 50000 samples, validate on 10000 samples
Epoch 1/5
50000/50000 [==============================] - 9s 188us/step - loss: 1.3676 -
acc: 0.5132 - val_loss: 1.6972 - val_acc: 0.4301
Epoch 2/5
50000/50000 [==============================] - 9s 182us/step - loss: 1.2194 -
acc: 0.5697 - val_loss: 1.3140 - val_acc: 0.5243
Epoch 3/5
50000/50000 [==============================] - 9s 183us/step - loss: 1.0931 -
acc: 0.6149 - val_loss: 1.4077 - val_acc: 0.5308
Epoch 4/5
50000/50000 [==============================] - 9s 183us/step - loss: 0.9930 -
acc: 0.6531 - val_loss: 1.2044 - val_acc: 0.5719
Epoch 5/5
50000/50000 [==============================] - 9s 183us/step - loss: 0.9001 -
acc: 0.6861 - val_loss: 1.0029 - val_acc: 0.6515
```

Out[13]: <keras.callbacks.History at 0x7f0599c885f8>

One of the more powerful features of the `fit()` method is the `callbacks` argument. We can use [prebuilt classes (https://keras.io/callbacks/)](https://keras.io/callbacks/), or create our own, that are called after every batch and epoch to update status or cause the fit to terminate. For example, we can use the [EarlyStopping (https://keras.io/callbacks/#earlystopping)](https://keras.io/callbacks/#earlystopping) to end the fit if no improvement larger than 5% is seen for 2 training epochs

In [14]:
```
early_stop = keras.callbacks.EarlyStopping(monitor='val_acc', min_delta=0.05,
patience=2, verbose=1)
model.fit(x_train, y_train,
          batch_size=256,
          epochs=10,
          verbose=1,
          validation_data=(x_test, y_test),
          callbacks=[early_stop])
```

```
Train on 50000 samples, validate on 10000 samples
Epoch 1/10
50000/50000 [==============================] - 9s 184us/step - loss: 0.8128 -
acc: 0.7169 - val_loss: 0.9697 - val_acc: 0.6701
Epoch 2/10
50000/50000 [==============================] - 9s 183us/step - loss: 0.7403 -
acc: 0.7428 - val_loss: 0.9699 - val_acc: 0.6728
Epoch 3/10
50000/50000 [==============================] - 9s 183us/step - loss: 0.6713 -
acc: 0.7668 - val_loss: 1.0208 - val_acc: 0.6510
Epoch 00003: early stopping
```

Out[14]:
```
<keras.callbacks.History at 0x7f0580374908>
```

Inspecting the Fit

Now that the model is trained, we can use the model object in various ways. First, we can look at the training history object:

In [16]:
```
print(history.epoch)
history.history
```

```
[0, 1, 2, 3, 4]
```

Out[16]:
```
{'acc': [0.2492000000190735,
  0.4271599999904633,
  0.5078199999904632,
  0.5670800000190734,
  0.6137600000190735],
 'loss': [2.069864055786133,
  1.6068210692977904,
  1.3832057107925415,
  1.2233458753204345,
  1.1020074940490723],
 'val_acc': [0.3842, 0.4268, 0.4865, 0.5531, 0.5997],
 'val_loss': [1.7479137044906616,
  1.5601007862091065,
  1.4178311756134032,
  1.2407939823150635,
  1.1233568576812745]}
```

The `history.history` dictionary has tracked several different values through the training process:

- `acc`: Accuracy of the model on the training set, averaged over the batches
- `val_acc`: Accuracy of the model on the validation set
- `loss`: Value of the loss function on the training set, averaged over the batches
- `val_loss`: Value of the loss function on the validation set

Note that the loss function on the training data is the only thing the optimizer is trying to minimize. The other metrics hopefully improve at the same time, but do not always.

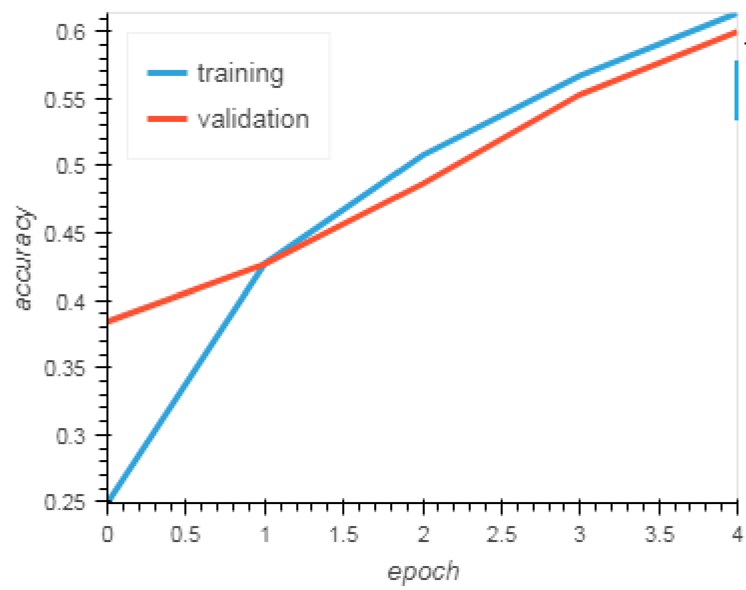
We can plot the accuracy on the test and training data with Holoviews:

In [17]:
```
%%opts Curve [width=400 height=300]
%%opts Curve (line_width=3)
%%opts Overlay [legend_position='top_left']

train_acc = hv.Curve((history.epoch, history.history['acc']), 'epoch', 'accura
cy', label='training')
val_acc = hv.Curve((history.epoch, history.history['val_acc']), 'epoch', 'accu
racy', label='validation')

train_acc * val_acc
```

Out[17]:



(htt

We can also look individual predictions. Let's run the final trained model over the validation set:

In [18]:
```
y_predict = model.predict(x_test)
y_predict[:5]
```

Out[18]:
```
array([[3.6380466e-02, 2.8012809e-03, 5.6521245e-03, 4.6179500e-01,
        1.8767768e-04, 4.1285092e-01, 6.5586432e-03, 2.9424774e-03,
        6.5226592e-02, 5.6048380e-03],
       [7.7234273e-04, 1.3481379e-04, 9.0244555e-07, 1.2523625e-08,
        1.3514633e-09, 2.3074564e-09, 2.2217771e-10, 2.4492710e-09,
        9.9903846e-01, 5.3437961e-05],
       [1.1238730e-01, 6.4417180e-03, 2.8757221e-04, 1.5218179e-04,
        9.8057908e-06, 8.1753742e-06, 2.7492351e-06, 1.5617818e-04,
        8.6178100e-01, 1.8773323e-02],
       [9.6699768e-01, 1.2896356e-02, 7.1832794e-03, 1.8573498e-03,
        2.6031092e-04, 2.1751088e-05, 7.5305172e-05, 5.0229981e-05,
        8.9624766e-03, 1.6951126e-03],
       [9.6883459e-06, 1.4818322e-04, 1.8114038e-02, 5.3014785e-02,
        8.4939939e-01, 1.6289981e-02, 6.2562138e-02, 9.9133758e-05,
        2.8525974e-04, 7.7422737e-05]], dtype=float32)
```

This is still using the one-hot encoding, where each input image produces 10 columns (for categories 0-9) of output. Normally, we would take the column with the largest output as the predicted category. We could do this with some NumPy magic, but Keras also includes a convenience method `predict_classes()`, which does this automatically:

In [19]:
```
y_predict = model.predict_classes(x_test)
y_predict[:5]
```

Out[19]:
```
array([3, 8, 8, 0, 4])
```

And then we can use our label array and NumPy fancy indexing to see these as strings:

In [22]:
```
y_predict_labels = cifar10_labels[y_predict]
y_true_labels = cifar10_labels[y_test_true]
print(y_predict_labels[:5])
print(y_true_labels[:5])
```

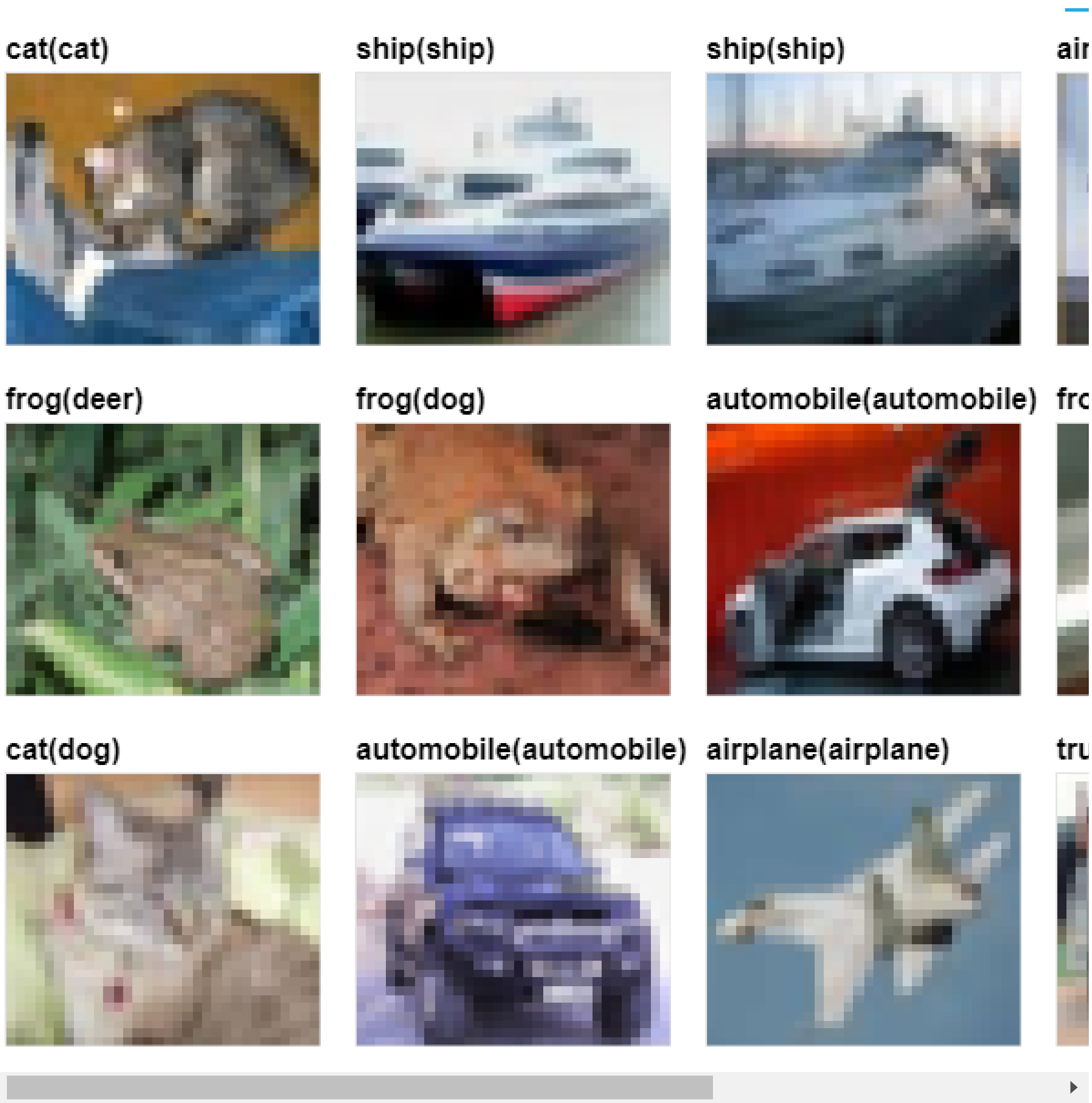
```
['cat' 'ship' 'ship' 'airplane' 'deer']
['cat' 'ship' 'ship' 'airplane' 'frog']
```

Holoviews makes it easy to look at the first few predictions:

In [23]:
```
%%output size=64
%%opts RGB [xaxis=None yaxis=None]

images = [hv.RGB(x_test[i], label='%s(%s)' % (y_true_labels[i], y_predict_labe
ls[i]) ) for i in range(12)]
hv.Layout(images).cols(4)
```

Out[23]:



In fact, let's select out the failed predictions with more NumPy fancy indexing:

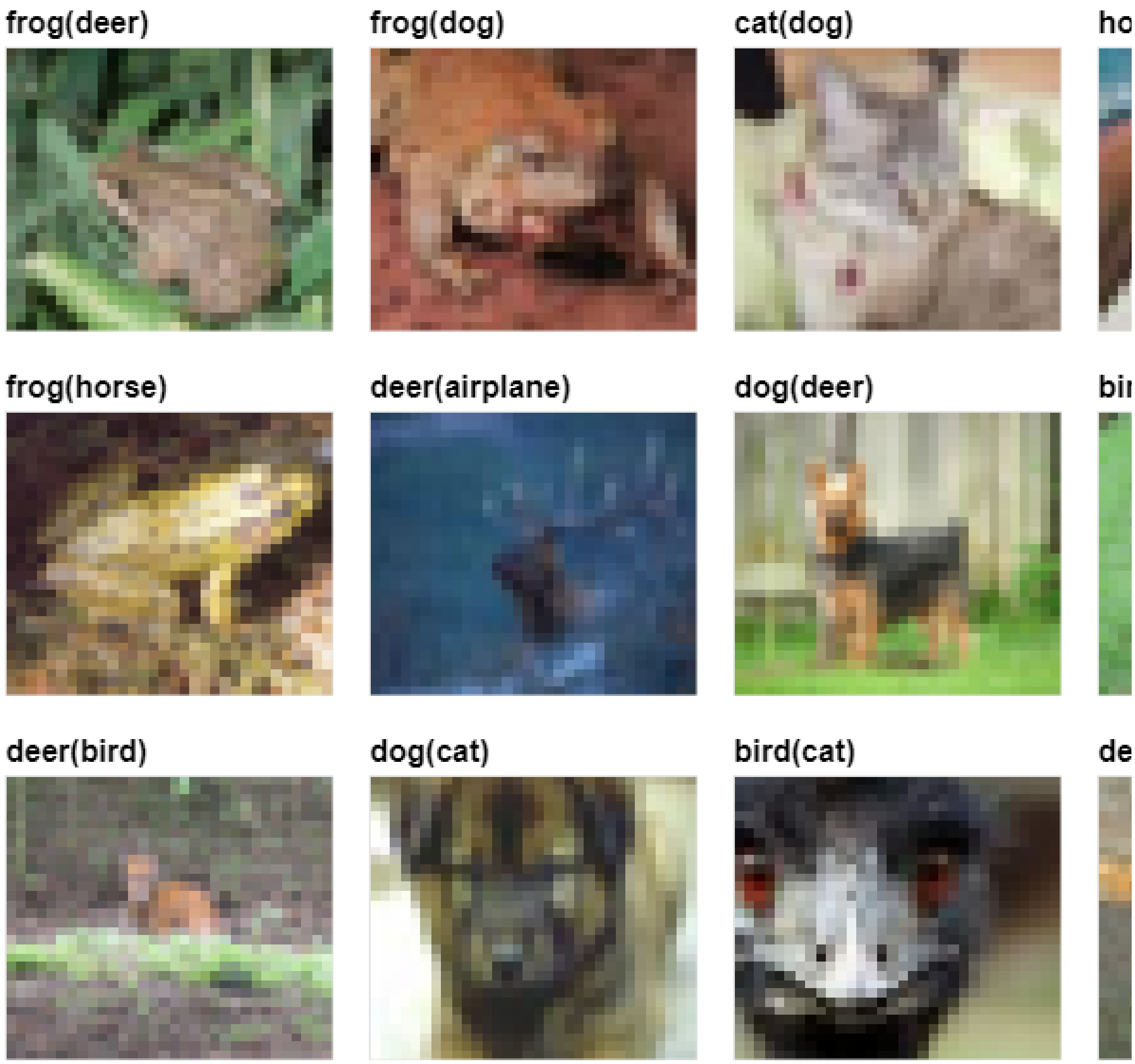
In [24]:
```
failed = y_predict != y_test_true
print('Number failed:', np.count_nonzero(failed))
```

```
Number failed: 3490
```

In [25]:
```
%%output size=64
%%opts RGB [xaxis=None yaxis=None]

images = [hv.RGB(x_test[failed][i], label='%s(%s)' %
                 (y_true_labels[failed][i],
                  y_predict_labels[failed][i]) ) for i in range(12)]
hv.Layout(images).cols(4)
```

Out[25]:



We'll learn more about evaluating the model in the next section.

Experiments to Try

- Try changing some of the model parameters (number of dense nodes, number of convolution kernels) and see how training changes.
- Try changing the batch size during training to see how the speed of training is affected (and the final accuracy).
- Try changing `relu` to `sigmoid`.

If you screw everything up, you can use File / Revert to Checkpoint to go back to the first version of the notebook and restart the Jupyter kernel with Kernel / Restart.

In [27]:
```
model = Sequential()

### Convolution and max pool layers

# Group 1: Convolution
model.add(Conv2D(32, kernel_size=(3, 3),
                 activation='sigmoid',
                 input_shape=x_train.shape[1:]))
model.add(Conv2D(32, (3, 3), activation='sigmoid'))
model.add(MaxPooling2D(pool_size=(2, 2)))

# Group 2: Convolution
model.add(Conv2D(64, kernel_size=(3, 3),
                 activation='sigmoid',
                 input_shape=x_train.shape[1:]))
model.add(Conv2D(64, (3, 3), activation='sigmoid'))
model.add(MaxPooling2D(pool_size=(2, 2)))

# Group 3: Dense layers
model.add(Flatten())
model.add(Dense(128, activation='sigmoid'))
model.add(Dense(num_classes, activation='softmax'))

model.compile(loss=keras.losses.categorical_crossentropy,
              optimizer=keras.optimizers.Adadelta(),
              metrics=['accuracy'])
```

In [28]:
```
model.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
conv2d_13 (Conv2D)           (None, 30, 30, 32)        896       
_________________________________________________________________
conv2d_14 (Conv2D)           (None, 28, 28, 32)        9248      
_________________________________________________________________
max_pooling2d_7 (MaxPooling2 (None, 14, 14, 32)        0         
_________________________________________________________________
conv2d_15 (Conv2D)           (None, 12, 12, 64)        18496     
_________________________________________________________________
conv2d_16 (Conv2D)           (None, 10, 10, 64)        36928     
_________________________________________________________________
max_pooling2d_8 (MaxPooling2 (None, 5, 5, 64)          0         
_________________________________________________________________
flatten_4 (Flatten)          (None, 1600)              0         
_________________________________________________________________
dense_7 (Dense)              (None, 128)               204928    
_________________________________________________________________
dense_8 (Dense)              (None, 10)                1290      
=================================================================
Total params: 271,786
Trainable params: 271,786
Non-trainable params: 0
_________________________________________________________________
```

In [29]:
```
%%time
history = model.fit(x_train, y_train,
          batch_size=256,
          epochs=5,
          verbose=1,
          validation_data=(x_test, y_test))
```

```
Train on 50000 samples, validate on 10000 samples
Epoch 1/5
50000/50000 [==============================] - 10s 192us/step - loss: 2.3147
- acc: 0.1007 - val_loss: 2.3503 - val_acc: 0.1000
Epoch 2/5
50000/50000 [==============================] - 9s 185us/step - loss: 2.3077 -
acc: 0.0995 - val_loss: 2.3065 - val_acc: 0.1000
Epoch 3/5
50000/50000 [==============================] - 9s 183us/step - loss: 2.3064 -
acc: 0.0981 - val_loss: 2.3083 - val_acc: 0.1000
Epoch 4/5
50000/50000 [==============================] - 9s 183us/step - loss: 2.3066 -
acc: 0.0990 - val_loss: 2.3197 - val_acc: 0.1000
Epoch 5/5
50000/50000 [==============================] - 9s 182us/step - loss: 2.3070 -
acc: 0.1000 - val_loss: 2.3067 - val_acc: 0.1000
CPU times: user 33.7 s, sys: 6.9 s, total: 40.6 s
Wall time: 46.6 s
```

In [31]:
```
model.fit(x_train, y_train,
          batch_size=128,    # changing the batch size to 128
          epochs=2,
          verbose=1,
          validation_data=(x_test, y_test))
```

```
Train on 50000 samples, validate on 10000 samples
Epoch 1/2
50000/50000 [==============================] - 11s 223us/step - loss: 2.3085
- acc: 0.0981 - val_loss: 2.3066 - val_acc: 0.1000
Epoch 2/2
50000/50000 [==============================] - 11s 219us/step - loss: 2.3079
- acc: 0.1000 - val_loss: 2.3084 - val_acc: 0.1000
```

Out[31]: <keras.callbacks.History at 0x7f0580371c88>

In [32]:
```
model.fit(x_train, y_train,
          batch_size=128,
          epochs=5,           # changing the epochs to see different values
          verbose=1,
          validation_data=(x_test, y_test))
```

```
Train on 50000 samples, validate on 10000 samples
Epoch 1/5
50000/50000 [==============================] - 11s 221us/step - loss: 2.2927
- acc: 0.1152 - val_loss: 2.1529 - val_acc: 0.2114
Epoch 2/5
50000/50000 [==============================] - 11s 219us/step - loss: 2.0657
- acc: 0.2489 - val_loss: 2.0555 - val_acc: 0.2553
Epoch 3/5
50000/50000 [==============================] - 11s 219us/step - loss: 1.9725
- acc: 0.2794 - val_loss: 1.9320 - val_acc: 0.3030
Epoch 4/5
50000/50000 [==============================] - 11s 219us/step - loss: 1.9064
- acc: 0.3035 - val_loss: 1.9348 - val_acc: 0.2879
Epoch 5/5
50000/50000 [==============================] - 11s 218us/step - loss: 1.8371
- acc: 0.3271 - val_loss: 1.7965 - val_acc: 0.3450
```

Out[32]: <keras.callbacks.History at 0x7f0580371e80>

In [ ]:
```
early_stop = keras.callbacks.EarlyStopping(monitor='val_acc', min_delta=0.05,
patience=2, verbose=1)
model.fit(x_train, y_train,
          batch_size=128,
          epochs=10,
          verbose=1,
          validation_data=(x_test, y_test),
          callbacks=[early_stop])
```

In [33]:
```
print(history.epoch)
history.history
```

```
[0, 1, 2, 3, 4]
```

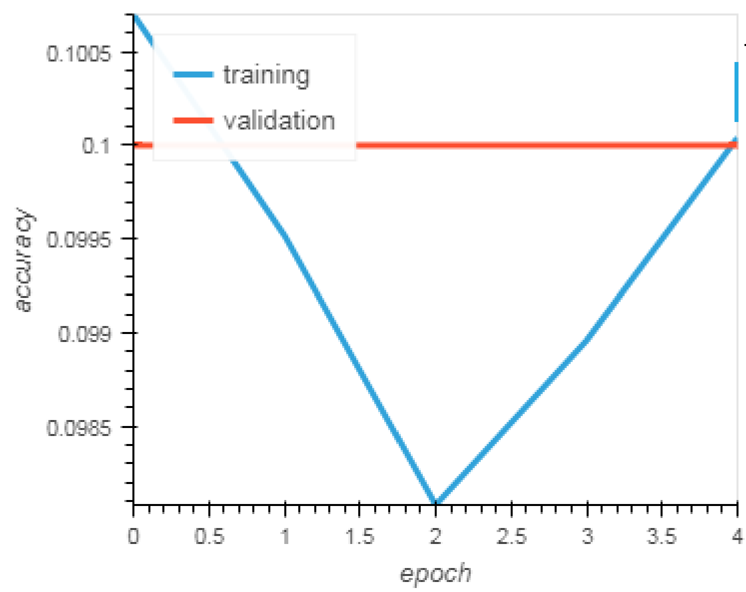
Out[33]:
```
{'acc': [0.10070000000953674,
  0.09952000000476838,
  0.09808000000476837,
  0.09895999999046326,
  0.10003999999523162],
 'loss': [2.3146527812957762,
  2.3076702716827393,
  2.3063686571502684,
  2.306552299194336,
  2.3069844996643067],
 'val_acc': [0.1, 0.1, 0.1, 0.1, 0.1],
 'val_loss': [2.3502677799224854,
  2.3065195571899415,
  2.3082505477905273,
  2.3197000480651857,
  2.3067420444488524]}
```

In [34]:
```
%%opts Curve [width=400 height=300]
%%opts Curve (line_width=3)
%%opts Overlay [legend_position='top_left']

train_acc = hv.Curve((history.epoch, history.history['acc']), 'epoch', 'accura
cy', label='training')
val_acc = hv.Curve((history.epoch, history.history['val_acc']), 'epoch', 'accu
racy', label='validation')

train_acc * val_acc
```

Out[34]:



(htt

In [35]:
```
y_predict = model.predict(x_test)
y_predict[:5]
```

Out[35]:
```
array([[0.05727238, 0.0284339 , 0.05304283, 0.31944028, 0.10451485,
        0.16078496, 0.06803529, 0.09591644, 0.10964811, 0.00291096],
       [0.17486767, 0.25367653, 0.00737436, 0.0056479 , 0.00686585,
        0.00347859, 0.00130694, 0.01194562, 0.31297526, 0.22186132],
       [0.25909007, 0.21031818, 0.01937399, 0.03228152, 0.01433384,
        0.02572134, 0.00473468, 0.01957722, 0.31784618, 0.09672301],
       [0.331518  , 0.18937214, 0.03052554, 0.03110795, 0.04006365,
        0.01402636, 0.00660338, 0.02156444, 0.299135  , 0.03608354],
       [0.01454755, 0.00592878, 0.13832147, 0.2228402 , 0.23589778,
        0.13578102, 0.13704802, 0.0892829 , 0.00959195, 0.0107604 ]],
      dtype=float32)
```

In [36]:
```
y_predict = model.predict_classes(x_test)
y_predict[:5]
```

Out[36]:
```
array([3, 8, 8, 0, 4])
```

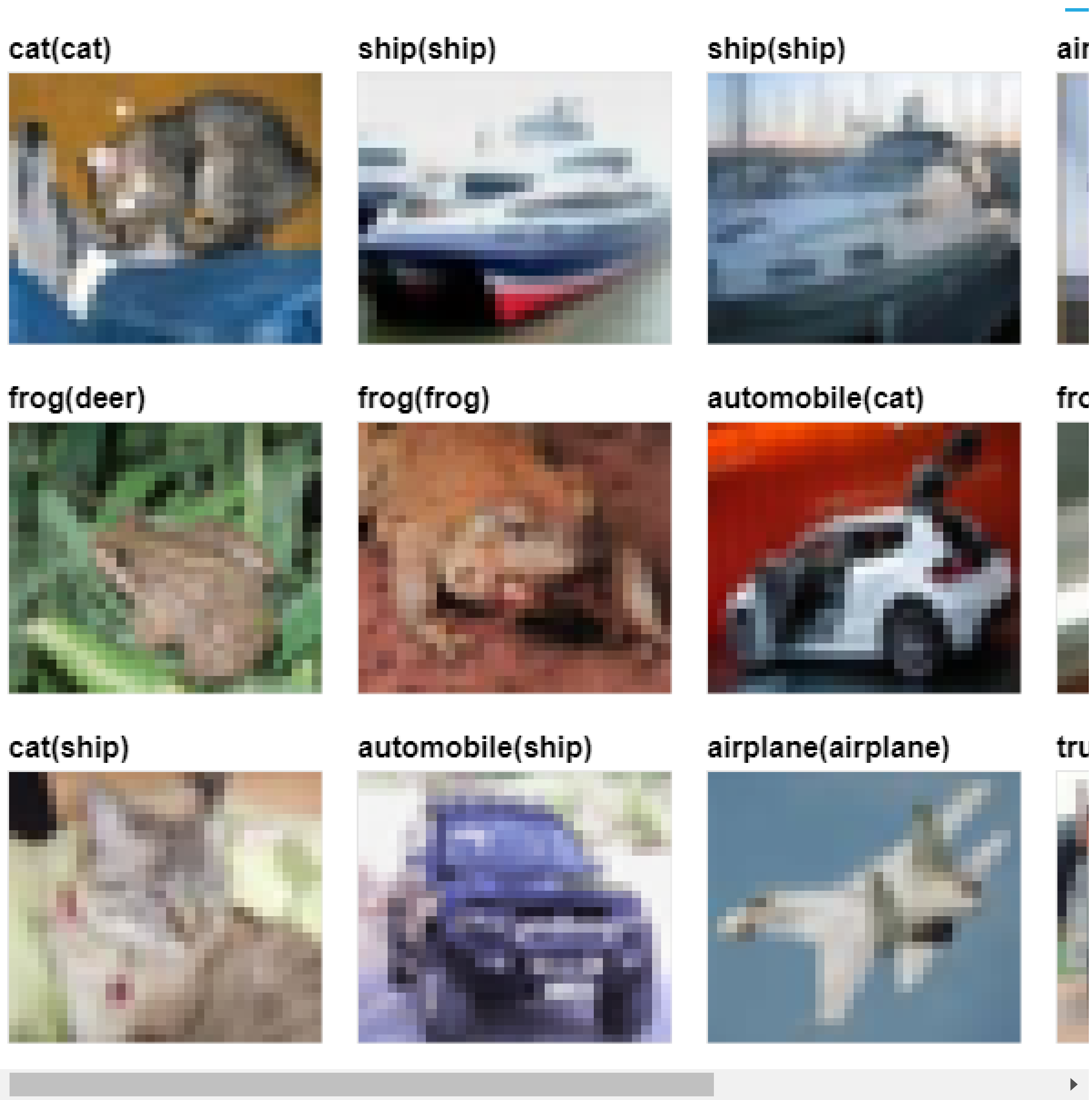
In [37]:
```
y_predict_labels = cifar10_labels[y_predict]
y_true_labels = cifar10_labels[y_test_true]
print(y_predict_labels[:5])
print(y_true_labels[:5])
```

```
['cat' 'ship' 'ship' 'airplane' 'deer']
['cat' 'ship' 'ship' 'airplane' 'frog']
```

In [38]:
```
%%output size=64
%%opts RGB [xaxis=None yaxis=None]

images = [hv.RGB(x_test[i], label='%s(%s)' % (y_true_labels[i], y_predict_labe
ls[i]) ) for i in range(12)]
hv.Layout(images).cols(4)
```

Out[38]:



In [39]:
```
failed = y_predict != y_test_true
print('Number failed:', np.count_nonzero(failed))
```

```
Number failed: 6550
```

In [40]:
```
%%output size=64
%%opts RGB [xaxis=None yaxis=None]

images = [hv.RGB(x_test[failed][i], label='%s(%s)' %
                  (y_true_labels[failed][i],
                   y_predict_labels[failed][i]) ) for i in range(12)]
hv.Layout(images).cols(4)
```

Out[40]: