REI602M Machine Learning - Homework 10

Due: *Thursday* 28.3.2019

Objectives: Convolutional neural networks (CNNs), Recurrent neural networks (RNNs)

Name: Emil Gauti Friðriksson, **email:** egf3@hi.is, **collaborators:** (if any)

Notes: You need TensorFlow for this assignment. Refer to installation instructions on Piazza.

1. [Image classification with CNNs, 50 points] The CIFAR-10 is a small image classification dataset consisting of 60000 color images of size 32x32 in 10 classes.
Human accuracy on CIFAR-10 is approximately 94% while state of the art CNNs achieve around 99% accuracy! You can expect accuracy close to 80% in this problem and around 84% in problem 2 (this is comparable to state of the art performance in 2013).

The data is split into training and validation sets below. There is no separate test set so the accuracy estimates that you obtain will be somewhat optimistic. Starting from a simple network architecture you gradually add layers, with the aim of improving accuracy. In each of the tasks below, you report the final training and validation accuracies and provide a graph showing how they change during training. What can you conclude from the graph in each case? Monitor the accuracy during training and stop when the validation accuracy no longer improves. Ten epochs should be sufficient in most cases.

In the following, INPUT denotes the input layer, FC denotes a fully connected layer, CONV-m represents a 2D-convolutional layer with m filters, POOL corresponds to a 2D pooling layer, RELU to ReLU activation units, [...]*n denotes repetition n times of the units inside the brackets. The last last layer (FCS) denotes a fully connected layer with 10 nodes and softmax activation (this is the classification step). Use dropout for regulatization and only following FC layers.

a) INPUT -> [FC -> RELU] -> FCS (conventional feedforward network)

b) INPUT -> [CONV-32 -> RELU -> POOL] -> FCS (minimalistic CNN)

c) INPUT -> [CONV-32 -> RELU]*2 -> POOL]*2 -> [FC -> RELU]*1 -> FCS

d) [CONV-32 -> RELU]*2 -> POOL -> [CONV-64 -> RELU]*2 -> POOL -> [CONV-128 -> RELU]*3 -> POOL -> FC -> FCS (simplified VGGnet)

Comments:

- Implement your networks using Keras. You can see examples of fully connected networks in `v10_nn_keras.ipynb` and a convolutional network in `v11_dnn.ipynb`.
- Regularization of convolutional layers does not seem to be very effective. Fully connected need regularization to prevent overfitting. Dropout with $p = 0.5$ is usually quite effective.
- Use `padding="same"` to zero-pad the input to convolutional layers.
- You can continue training a model by calling `model.fit` repeatedly.
- To save a model use `model.save(filename)`. You may also want to look into model checkpoints and early stopping. See `ModelCheckpoint` and `EarlyStopping` in the Keras documentation.
- The CIFAR-10 "high score" was obtained by training giant deep networks on huge image databases in order to learn feature maps relevant to image classification. The networks were

then fine-tuned on CIFAR-10 (this an example of *transfer learning*).

In [4]:

```
# Load the CIFAR-10 data set of tiny images (~170 MB)
import numpy as np
from tensorflow import keras
(x_train, y_train), (x_test, y_test) = keras.datasets.cifar10.load_data() # Takes c
onsiderable time first time around

print('x_train shape:', x_train.shape)
print(x_train.shape[0], 'train samples')
print(x_test.shape[0], 'test samples')

# Convert class vectors to binary class matrices.
num_classes=len(np.unique(y_train))
print("Number of classes:", num_classes)
y_train = keras.utils.to_categorical(y_train, num_classes)
y_test = keras.utils.to_categorical(y_test, num_classes)

# Convert to 32-bit floats
x_train = x_train.astype('float32')
x_test = x_test.astype('float32')

# Scale data
x_train /= 255
x_test /= 255

# THINK:
x_val = x_test
y_val = y_test
```

```
x_train shape: (50000, 32, 32, 3)
50000 train samples
10000 test samples
Number of classes: 10
```

I let all the tests run for 10 epochs as the time it takes isn't that much

In [19]:

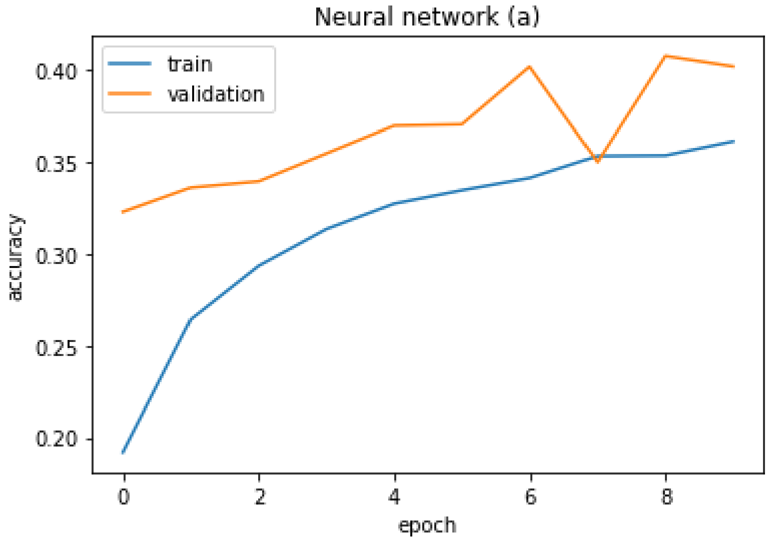
```
#a) INPUT -> [FC -> RELU] -> FCS (conventional feedforward network)
from tensorflow.python.keras.models import Sequential
from tensorflow.python.keras.layers import Dropout
from tensorflow.python.keras.layers import Dense#, Activation
from tensorflow.python.keras.layers import Flatten
from tensorflow.python.keras.layers import Conv2D, MaxPooling2D
from tensorflow.python.keras.optimizers import Adadelta
num_epochs = 10
batch_sizeA=300
input_shape = x_train.shape[1:]
modelA = Sequential()
modelA.add(Flatten(input_shape=input_shape))
modelA.add(Dense(300, activation='relu'))
modelA.add(Dropout(0.5))
modelA.add(Dense(num_classes, activation='softmax'))

modelA.summary()

modelA.compile(loss='categorical_crossentropy',
```

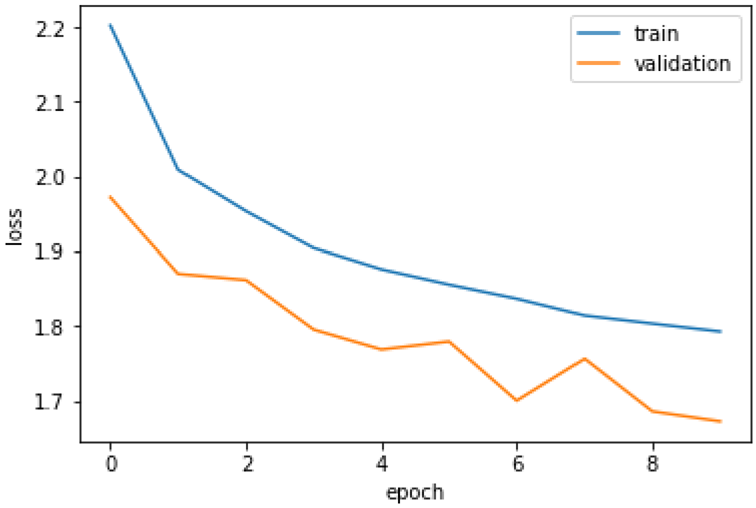
```
              optimizer=Adadelta(),
              metrics=['accuracy'])

historyA=modelA.fit(x_train, y_train,
                    batch_size=batch_sizeA,
                    epochs=num_epochs,
                    verbose=1,
                    validation_data=(x_val, y_val))
scoreA = modelA.evaluate(x_val, y_val, verbose=0)
print('val loss:', scoreA[0])
print('val accuracy:', scoreA[1])
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
flatten_13 (Flatten)         (None, 3072)              0         
_________________________________________________________________
dense_17 (Dense)             (None, 300)               921900    
_________________________________________________________________
dropout_3 (Dropout)          (None, 300)               0         
_________________________________________________________________
dense_18 (Dense)             (None, 10)                3010      
=================================================================
Total params: 924,910
Trainable params: 924,910
Non-trainable params: 0
_________________________________________________________________
Train on 50000 samples, validate on 10000 samples
Epoch 1/10
50000/50000 [==============================] - 3s 64us/step - loss: 2.
2026 - acc: 0.1921 - val_loss: 1.9722 - val_acc: 0.3228
Epoch 2/10
50000/50000 [==============================] - 2s 46us/step - loss: 2.
0091 - acc: 0.2643 - val_loss: 1.8699 - val_acc: 0.3360
Epoch 3/10
50000/50000 [==============================] - 2s 46us/step - loss: 1.
9537 - acc: 0.2934 - val_loss: 1.8614 - val_acc: 0.3394
Epoch 4/10
50000/50000 [==============================] - 2s 46us/step - loss: 1.
9043 - acc: 0.3134 - val_loss: 1.7952 - val_acc: 0.3545
Epoch 5/10
50000/50000 [==============================] - 2s 46us/step - loss: 1.
8758 - acc: 0.3272 - val_loss: 1.7686 - val_acc: 0.3696
Epoch 6/10
50000/50000 [==============================] - 2s 46us/step - loss: 1.
8554 - acc: 0.3345 - val_loss: 1.7793 - val_acc: 0.3703
Epoch 7/10
50000/50000 [==============================] - 2s 46us/step - loss: 1.
8366 - acc: 0.3412 - val_loss: 1.7000 - val_acc: 0.4016
Epoch 8/10
50000/50000 [==============================] - 2s 47us/step - loss: 1.
8141 - acc: 0.3531 - val_loss: 1.7562 - val_acc: 0.3498
Epoch 9/10
50000/50000 [==============================] - 2s 46us/step - loss: 1.
8033 - acc: 0.3534 - val_loss: 1.6855 - val_acc: 0.4074
Epoch 10/10
50000/50000 [==============================] - 2s 46us/step - loss: 1.
7927 - acc: 0.3611 - val_loss: 1.6722 - val_acc: 0.4017


val loss: 1.6721840606689453
val accuracy: 0.4017
```

In [25]:

```
#Plottum gögnin okkar
#(a)
import matplotlib.pyplot as plt
plt.plot(historyA.history['acc'])
plt.plot(historyA.history['val_acc'])
plt.ylabel('accuracy')
plt.xlabel('epoch')
plt.title('Neural network (a)')
plt.legend(['train', 'validation'])
plt.show()

plt.plot(historyA.history['loss'])
plt.plot(historyA.history['val_loss'])
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['train', 'validation'])
plt.show()
```





We are not getting great accuracy over here but this is only the start, we expect the accuracy as well as the loss function to return better values as the models get more complicated

In [5]:

```
#(b)INPUT -> [CONV-32 -> RELU -> POOL] -> FCS (minimalistic CNN)
```

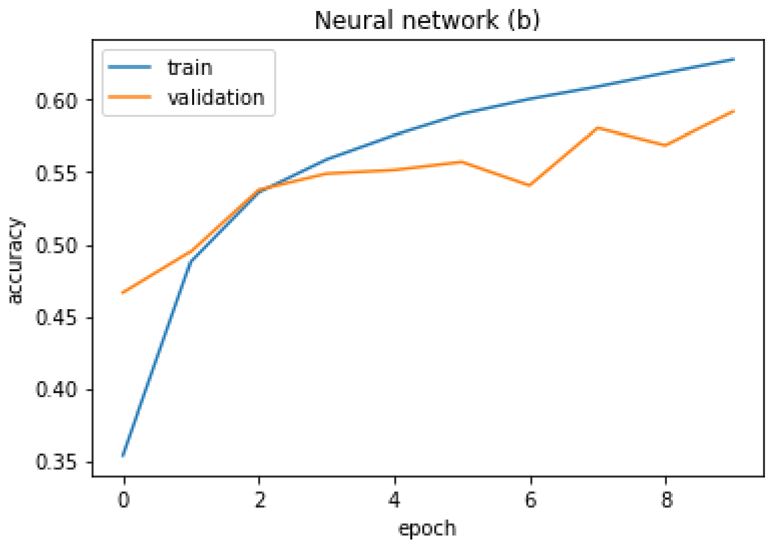
```python
from tensorflow.python.keras.models import Sequential
from tensorflow.python.keras.layers import Dropout
from tensorflow.python.keras.layers import Dense#, Activation
from tensorflow.python.keras.layers import Flatten
from tensorflow.python.keras.layers import Conv2D, MaxPooling2D
from tensorflow.python.keras.optimizers import Adadelta
batch_sizeB=300
num_epochs=10
input_shape = x_train.shape[1:]
modelB = Sequential()

modelB.add(Conv2D(32, kernel_size=(3, 3),
                 activation='relu',
                 input_shape=input_shape, padding = "same"))
modelB.add(MaxPooling2D(pool_size=(2, 2)))

modelB.add(Flatten())
modelB.add(Dense(10, activation='softmax'))


modelB.summary()

modelB.compile(loss='categorical_crossentropy',
              optimizer=Adadelta(),
              metrics=['accuracy'])

historyB=modelB.fit(x_train, y_train,
                  batch_size=batch_sizeB,
                  epochs=num_epochs,
                  verbose=1,
                  validation_data=(x_val, y_val))
scoreB = modelB.evaluate(x_val, y_val, verbose=0)
print('val loss:', scoreB[0])
print('val accuracy:', scoreB[1])
```

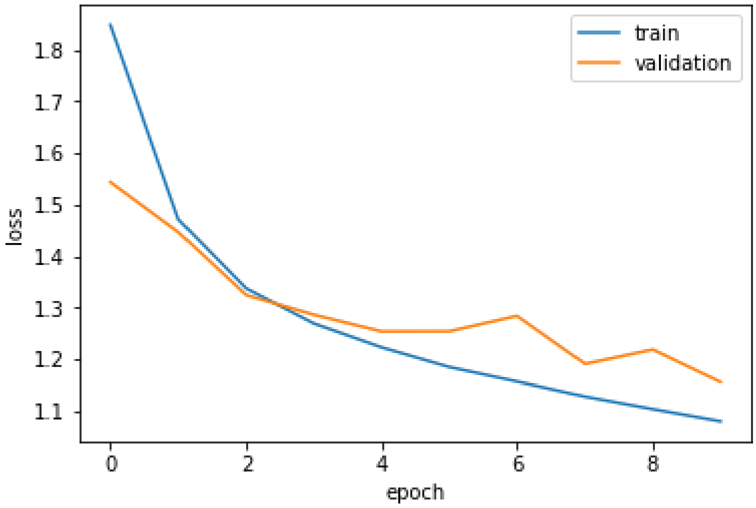
```
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
conv2d (Conv2D)              (None, 32, 32, 32)        896       
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 16, 16, 32)        0         
_________________________________________________________________
flatten (Flatten)            (None, 8192)              0         
_________________________________________________________________
dense_1 (Dense)              (None, 10)                81930     
=================================================================
Total params: 82,826
Trainable params: 82,826
Non-trainable params: 0
_________________________________________________________________
Train on 50000 samples, validate on 10000 samples
Epoch 1/10
50000/50000 [==============================] - 4s 85us/step - loss: 1.
8481 - acc: 0.3537 - val_loss: 1.5437 - val_acc: 0.4665
Epoch 2/10
50000/50000 [==============================] - 3s 54us/step - loss: 1.
4714 - acc: 0.4881 - val_loss: 1.4466 - val_acc: 0.4950
Epoch 3/10
50000/50000 [==============================] - 3s 54us/step - loss: 1.
3371 - acc: 0.5361 - val_loss: 1.3242 - val_acc: 0.5379
Epoch 4/10
50000/50000 [==============================] - 3s 54us/step - loss: 1
```

```
50000/50000 [==============================] - 3s 54us/step - loss: 1.
2688 - acc: 0.5589 - val_loss: 1.2859 - val_acc: 0.5491
Epoch 5/10
50000/50000 [==============================] - 3s 55us/step - loss: 1.
2225 - acc: 0.5757 - val_loss: 1.2535 - val_acc: 0.5516
Epoch 6/10
50000/50000 [==============================] - 3s 54us/step - loss: 1.
1844 - acc: 0.5904 - val_loss: 1.2537 - val_acc: 0.5572
Epoch 7/10
50000/50000 [==============================] - 3s 54us/step - loss: 1.
1565 - acc: 0.6006 - val_loss: 1.2836 - val_acc: 0.5409
Epoch 8/10
50000/50000 [==============================] - 3s 54us/step - loss: 1.
1273 - acc: 0.6092 - val_loss: 1.1906 - val_acc: 0.5806
Epoch 9/10
50000/50000 [==============================] - 3s 54us/step - loss: 1.
1031 - acc: 0.6188 - val_loss: 1.2183 - val_acc: 0.5687
Epoch 10/10
50000/50000 [==============================] - 3s 54us/step - loss: 1.
0796 - acc: 0.6281 - val_loss: 1.1555 - val_acc: 0.5920
val loss: 1.1554940679550172
val accuracy: 0.592
```

In [7]:

```python
#Plottum gögnin okkar
#(b)
import matplotlib.pyplot as plt
plt.plot(historyB.history['acc'])
plt.plot(historyB.history['val_acc'])
plt.ylabel('accuracy')
plt.xlabel('epoch')
plt.title('Neural network (b)')
plt.legend(['train', 'validation'])
plt.show()

plt.plot(historyB.history['loss'])
plt.plot(historyB.history['val_loss'])
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['train', 'validation'])
plt.show()
```





These results are a lot more promising than what we got from (a). In both the graphs we see how train and validation stay more tightly together than in (a). We also get more accuracy as well as a lower loss

In [16]:

```
#c) INPUT -> [[CONV-32 -> RELU]*2 -> POOL]*2 -> [FC -> RELU]*1 -> FCS
from tensorflow.python.keras.models import Sequential
from tensorflow.python.keras.layers import Dropout
from tensorflow.python.keras.layers import Dense#, Activation
from tensorflow.python.keras.layers import Flatten
from tensorflow.python.keras.layers import Conv2D, MaxPooling2D
from tensorflow.python.keras.optimizers import Adadelta
batch_sizeB=300
num_epochs=10
input_shape = x_train.shape[1:]
modelC = Sequential()

modelC.add(Conv2D(32, kernel_size=(3, 3),
                 activation='relu',
                 input_shape=input_shape, padding = "same"))
modelC.add(Conv2D(32, kernel_size=(3, 3),
                 activation='relu',
                 input_shape=input_shape, padding = "same"))

modelC.add(MaxPooling2D(pool_size=(2, 2)))

modelC.add(Conv2D(32, kernel_size=(3, 3),
                 activation='relu',
                 input_shape=input_shape, padding = "same"))
modelC.add(Conv2D(32, kernel_size=(3, 3),
                 activation='relu',
                 input_shape=input_shape, padding = "same"))

modelC.add(MaxPooling2D(pool_size=(2, 2)))

modelC.add(Flatten())
modelC.add(Dense(100, activation='relu'))
modelC.add(Dense(10, activation='softmax'))



modelC.summary()
```

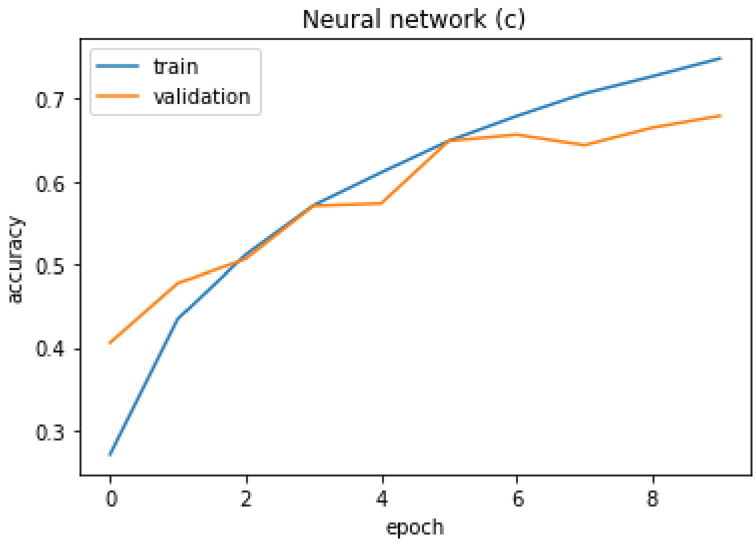
```
modelC.compile(loss='categorical_crossentropy',
              optimizer=Adadelta(),
              metrics=['accuracy'])

historyC=modelC.fit(x_train, y_train,
                  batch_size=batch_sizeB,
                  epochs=num_epochs,
                  verbose=1,
                  validation_data=(x_val, y_val))
scoreC = modelC.evaluate(x_val, y_val, verbose=0)
print('val loss:', scoreC[0])
print('val accuracy:', scoreC[1])
```

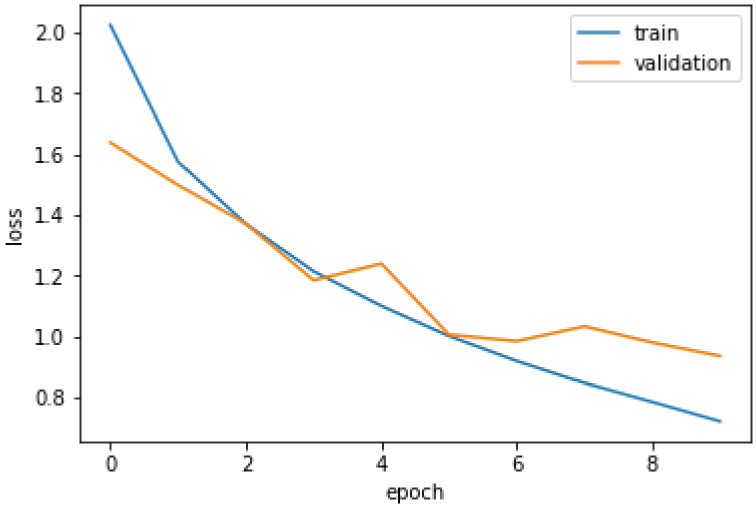
```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_31 (Conv2D)           (None, 32, 32, 32)        896
_________________________________________________________________
conv2d_32 (Conv2D)           (None, 32, 32, 32)        9248
_________________________________________________________________
max_pooling2d_17 (MaxPooling (None, 16, 16, 32)        0
_________________________________________________________________
conv2d_33 (Conv2D)           (None, 16, 16, 32)        9248
_________________________________________________________________
conv2d_34 (Conv2D)           (None, 16, 16, 32)        9248
_________________________________________________________________
max_pooling2d_18 (MaxPooling (None, 8, 8, 32)          0
_________________________________________________________________
flatten_11 (Flatten)         (None, 2048)              0
_________________________________________________________________
dense_13 (Dense)             (None, 100)               204900
_________________________________________________________________
dense_14 (Dense)             (None, 10)                1010
=================================================================
Total params: 234,550
Trainable params: 234,550
Non-trainable params: 0
_________________________________________________________________
Train on 50000 samples, validate on 10000 samples
Epoch 1/10
50000/50000 [==============================] - 5s 104us/step - loss:
2.0253 - acc: 0.2714 - val_loss: 1.6393 - val_acc: 0.4061
Epoch 2/10
50000/50000 [==============================] - 4s 87us/step - loss: 1.
5758 - acc: 0.4348 - val_loss: 1.4994 - val_acc: 0.4777
Epoch 3/10
50000/50000 [==============================] - 4s 86us/step - loss: 1.
3701 - acc: 0.5120 - val_loss: 1.3718 - val_acc: 0.5068
Epoch 4/10
50000/50000 [==============================] - 4s 86us/step - loss: 1.
2143 - acc: 0.5712 - val_loss: 1.1852 - val_acc: 0.5706
Epoch 5/10
50000/50000 [==============================] - 4s 85us/step - loss: 1.

1005 - acc: 0.6108 - val_loss: 1.2403 - val_acc: 0.5734
Epoch 6/10
50000/50000 [==============================] - 4s 86us/step - loss: 1.
0011 - acc: 0.6489 - val_loss: 1.0060 - val_acc: 0.6488
Epoch 7/10
50000/50000 [==============================] - 4s 86us/step - loss: 0.
9192 - acc: 0.6788 - val_loss: 0.9845 - val_acc: 0.6565
```

```
Epoch 8/10
50000/50000 [==============================] - 4s 87us/step - loss: 0.
8467 - acc: 0.7061 - val_loss: 1.0331 - val_acc: 0.6438
Epoch 9/10
50000/50000 [==============================] - 4s 87us/step - loss: 0.
7848 - acc: 0.7267 - val_loss: 0.9800 - val_acc: 0.6648
Epoch 10/10
50000/50000 [==============================] - 4s 86us/step - loss: 0.
7212 - acc: 0.7483 - val_loss: 0.9357 - val_acc: 0.6791
val loss: 0.9357429313659668
val accuracy: 0.6791
```

In [27]:

```
#Plottum gögnin okkar
#(c)
import matplotlib.pyplot as plt
plt.plot(historyC.history['acc'])
plt.plot(historyC.history['val_acc'])
plt.ylabel('accuracy')
plt.xlabel('epoch')
plt.title('Neural network (c)')
plt.legend(['train', 'validation'])
plt.show()

plt.plot(historyC.history['loss'])
plt.plot(historyC.history['val_loss'])
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['train', 'validation'])
plt.show()
```





These results are similar to the one we got in (b) but the accuracy has risen and the loss has gone down even further. This is so far the best results.

In [17]:

```
#[CONV-32->RELU]*2 -->POOL->[CONV-64 -> RELU]*2->POOL->[CONV-128 -> RELU]*3 -> POOL
-> FC -> FCS
from tensorflow.python.keras.models import Sequential
from tensorflow.python.keras.layers import Dropout
from tensorflow.python.keras.layers import Dense#, Activation
from tensorflow.python.keras.layers import Flatten
from tensorflow.python.keras.layers import Conv2D, MaxPooling2D
from tensorflow.python.keras.optimizers import Adadelta
batch_sizeD=300
num_epochs=10
input_shape = x_train.shape[1:]
modelD = Sequential()

modelD.add(Conv2D(32, kernel_size=(3, 3),
                 activation='relu',
                 input_shape=input_shape, padding = "same"))
modelD.add(Conv2D(32, kernel_size=(3, 3),
                 activation='relu',
                 input_shape=input_shape, padding = "same"))

modelD.add(MaxPooling2D(pool_size=(2, 2)))

modelD.add(Conv2D(64, kernel_size=(3, 3),
                 activation='relu',
                 input_shape=input_shape, padding = "same"))
modelD.add(Conv2D(64, kernel_size=(3, 3),
                 activation='relu',
                 input_shape=input_shape, padding = "same"))

modelD.add(MaxPooling2D(pool_size=(2, 2)))

modelD.add(Conv2D(128, kernel_size=(3, 3),
                 activation='relu',
                 input_shape=input_shape, padding = "same"))
modelD.add(Conv2D(128, kernel_size=(3, 3),
                 activation='relu',
                 input_shape=input_shape, padding = "same"))
```

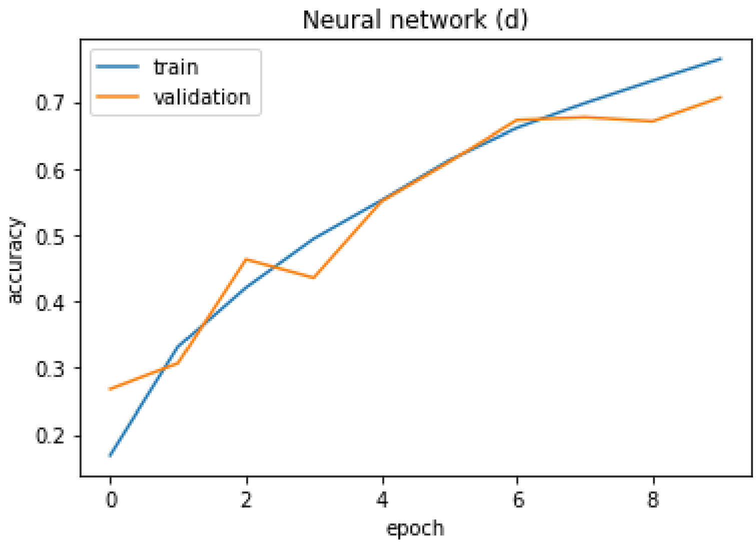
```
modelD.add(Conv2D(128, kernel_size=(3, 3),
                 activation='relu',
                 input_shape=input_shape, padding = "same"))

modelD.add(MaxPooling2D(pool_size=(2, 2)))

modelD.add(Flatten())
modelD.add(Dense(100, activation='relu'))
modelD.add(Dense(10, activation='softmax'))



modelD.summary()

modelD.compile(loss='categorical_crossentropy',
              optimizer=Adadelta(),
              metrics=['accuracy'])

historyD=modelD.fit(x_train, y_train,
                  batch_size=batch_sizeD,
                  epochs=num_epochs,
                  verbose=1,
                  validation_data=(x_val, y_val))
scoreD = modelD.evaluate(x_val, y_val, verbose=0)
print('val loss:', scoreD[0])
print('val accuracy:', scoreD[1])
```

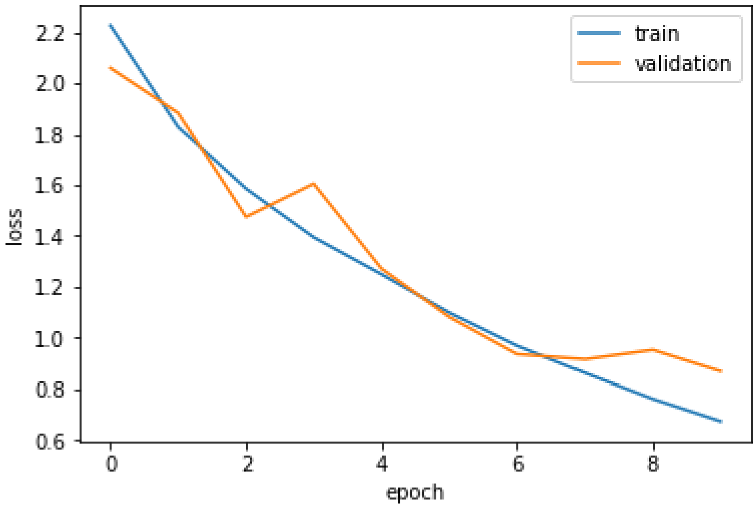
```
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
conv2d_35 (Conv2D)           (None, 32, 32, 32)        896       
_________________________________________________________________
conv2d_36 (Conv2D)           (None, 32, 32, 32)        9248      
_________________________________________________________________
max_pooling2d_19 (MaxPooling (None, 16, 16, 32)        0         
_________________________________________________________________
conv2d_37 (Conv2D)           (None, 16, 16, 64)        18496     
_________________________________________________________________
conv2d_38 (Conv2D)           (None, 16, 16, 64)        36928     
_________________________________________________________________
max_pooling2d_20 (MaxPooling (None, 8, 8, 64)          0         
_________________________________________________________________
conv2d_39 (Conv2D)           (None, 8, 8, 128)         73856     
_________________________________________________________________
conv2d_40 (Conv2D)           (None, 8, 8, 128)         147584    
_________________________________________________________________
conv2d_41 (Conv2D)           (None, 8, 8, 128)         147584    
_________________________________________________________________
max_pooling2d_21 (MaxPooling (None, 4, 4, 128)         0         
_________________________________________________________________
flatten_12 (Flatten)         (None, 2048)              0         
_________________________________________________________________
dense_15 (Dense)             (None, 100)               204900    
_________________________________________________________________
dense_16 (Dense)             (None, 10)                1010      
=================================================================
Total params: 640,502
Trainable params: 640,502
Non-trainable params: 0
_________________________________________________________________
Train on 50000 samples, validate on 10000 samples
Epoch 1/10
```

```
50000/50000 [==============================] - 8s 156us/step - loss:
2.2272 - acc: 0.1687 - val_loss: 2.0605 - val_acc: 0.2679
Epoch 2/10
50000/50000 [==============================] - 6s 128us/step - loss:
1.8294 - acc: 0.3316 - val_loss: 1.8840 - val_acc: 0.3067
Epoch 3/10
50000/50000 [==============================] - 6s 127us/step - loss:
1.5861 - acc: 0.4207 - val_loss: 1.4752 - val_acc: 0.4631
Epoch 4/10
50000/50000 [==============================] - 6s 127us/step - loss:
1.3947 - acc: 0.4942 - val_loss: 1.6051 - val_acc: 0.4353
Epoch 5/10
50000/50000 [==============================] - 6s 127us/step - loss:
1.2494 - acc: 0.5520 - val_loss: 1.2717 - val_acc: 0.5502
Epoch 6/10
50000/50000 [==============================] - 6s 127us/step - loss:
1.0986 - acc: 0.6123 - val_loss: 1.0815 - val_acc: 0.6096
Epoch 7/10
50000/50000 [==============================] - 6s 127us/step - loss:
0.9690 - acc: 0.6605 - val_loss: 0.9361 - val_acc: 0.6723
Epoch 8/10
50000/50000 [==============================] - 6s 128us/step - loss:
0.8626 - acc: 0.6977 - val_loss: 0.9162 - val_acc: 0.6766
Epoch 9/10
50000/50000 [==============================] - 6s 127us/step - loss:
0.7603 - acc: 0.7315 - val_loss: 0.9527 - val_acc: 0.6703
Epoch 10/10
50000/50000 [==============================] - 6s 126us/step - loss:
0.6731 - acc: 0.7641 - val_loss: 0.8695 - val_acc: 0.7062
val loss: 0.8694877672195435
val accuracy: 0.7062
```

In [28]:

```
#Plottum gögnin okkar
#(d)
import matplotlib.pyplot as plt
plt.plot(historyD.history['acc'])
plt.plot(historyD.history['val_acc'])
plt.ylabel('accuracy')
plt.xlabel('epoch')
plt.title('Neural network (d)')
plt.legend(['train', 'validation'])
plt.show()

plt.plot(historyD.history['loss'])
plt.plot(historyD.history['val_loss'])
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['train', 'validation'])
plt.show()
```





The accuracy and loss of the validation follow the trendline of the train very closely and intersect it often. This is a new behaviour. This is also the best results in regards to accuracy and loss.

2. [Data augmentation, 20 points] When the amount of training data is small in relation to the number of parameters in a model, overfitting becomes an issue. In many specialized image recognition tasks such as tumor classification, the amount of labeled data is often quite limited and a state of the art convolutional network are likely to severly overfit the data set. Data augmentation refers to techniques that create additional training examples from the original data set. For image data it is possible to create additional training examples by simple operations such as reflection, cropping and translation as well as by changing the color palette.

Take the best network from problem 1) and perform image augmention *during* training using the `ImageDataGenerator` class in Keras. You may need to train for more than 10 epochs. Report your results in the same way as you did in problem 1). Comment briefly on the type of mistakes that your network makes.

We will be using model D as it gave the best results. We'll be increasing the number of epochs to 20.

In [11]:

```
from tensorflow.python.keras.models import Sequential
from tensorflow.python.keras.preprocessing.image import ImageDataGenerator
from tensorflow.python.keras.layers import Dropout
```

```
from tensorflow.python.keras.layers import Dense#, Activation
from tensorflow.python.keras.layers import Flatten
from tensorflow.python.keras.layers import Conv2D, MaxPooling2D
from tensorflow.python.keras.optimizers import Adadelta
batch_size2=300
num_epochs2=20
input_shape = x_train.shape[1:]

model2 = Sequential()

model2.add(Conv2D(32, kernel_size=(3, 3),
                 activation='relu',
                 input_shape=input_shape, padding = "same"))
model2.add(Conv2D(32, kernel_size=(3, 3),
                 activation='relu',
                 input_shape=input_shape, padding = "same"))

model2.add(MaxPooling2D(pool_size=(2, 2)))

model2.add(Conv2D(64, kernel_size=(3, 3),
                 activation='relu',
                 input_shape=input_shape, padding = "same"))
model2.add(Conv2D(64, kernel_size=(3, 3),
                 activation='relu',
                 input_shape=input_shape, padding = "same"))

model2.add(MaxPooling2D(pool_size=(2, 2)))

model2.add(Conv2D(128, kernel_size=(3, 3),
                 activation='relu',
                 input_shape=input_shape, padding = "same"))
model2.add(Conv2D(128, kernel_size=(3, 3),
                 activation='relu',
                 input_shape=input_shape, padding = "same"))
model2.add(Conv2D(128, kernel_size=(3, 3),
                 activation='relu',
                 input_shape=input_shape, padding = "same"))

model2.add(MaxPooling2D(pool_size=(2, 2)))

model2.add(Flatten())
model2.add(Dense(100, activation='relu'))
model2.add(Dense(10, activation='softmax'))



model2.summary()


model2.compile(loss='categorical_crossentropy',
              optimizer=Adadelta(),
              metrics=['accuracy'])

datagen = ImageDataGenerator(featurewise_center=False, samplewise_center=False,
                             featurewise_std_normalization=False,
                             samplewise_std_normalization=False,
                             zca_whitening=False, rotation_range=0,
                             width_shift_range=0.1, height_shift_range=0.1,
                             horizontal_flip = False, vertical_flip = True)

datagen.fit(x_train)
```

```
history2 = model2.fit_generator(datagen.flow(x_train, y_train, batch_size=batch_siz
e2),
                              steps_per_epoch=len(x_train) / batch_size2,
                              epochs=num_epochs2,validation_data=(x_val, y_val))


score2 = model2.evaluate(x_val, y_val, verbose=0)
print('val loss:', score2[0])
print('val accuracy:', score2[1])
```

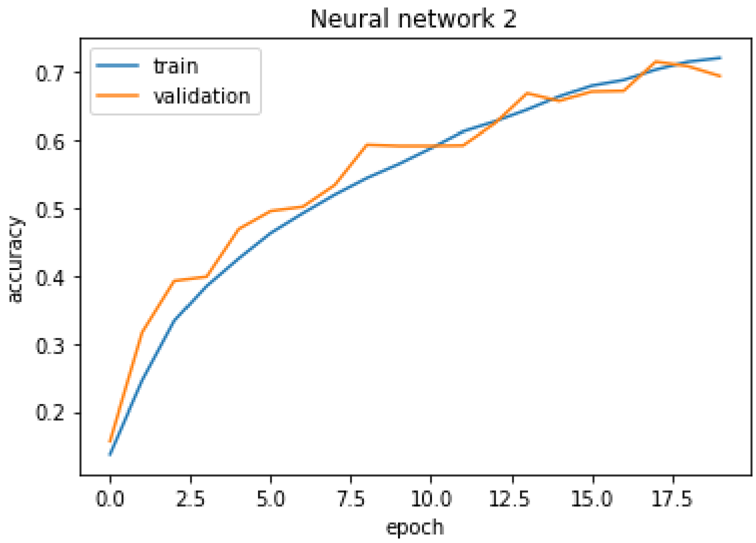
```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_8 (Conv2D)            (None, 32, 32, 32)        896
_________________________________________________________________
conv2d_9 (Conv2D)            (None, 32, 32, 32)        9248
_________________________________________________________________
max_pooling2d_4 (MaxPooling2 (None, 16, 16, 32)        0
_________________________________________________________________
conv2d_10 (Conv2D)           (None, 16, 16, 64)        18496
_________________________________________________________________
conv2d_11 (Conv2D)           (None, 16, 16, 64)        36928
_________________________________________________________________
max_pooling2d_5 (MaxPooling2 (None, 8, 8, 64)          0
_________________________________________________________________
conv2d_12 (Conv2D)           (None, 8, 8, 128)         73856
_________________________________________________________________
conv2d_13 (Conv2D)           (None, 8, 8, 128)         147584
_________________________________________________________________
conv2d_14 (Conv2D)           (None, 8, 8, 128)         147584
_________________________________________________________________
max_pooling2d_6 (MaxPooling2 (None, 4, 4, 128)         0
_________________________________________________________________
flatten_2 (Flatten)          (None, 2048)              0
_________________________________________________________________
dense_4 (Dense)              (None, 100)               204900
_________________________________________________________________
dense_5 (Dense)              (None, 10)                1010
=================================================================
Total params: 640,502
Trainable params: 640,502
Non-trainable params: 0
_________________________________________________________________
Epoch 1/20
167/166 [==============================] - 20s 122ms/step - loss: 2.23
78 - acc: 0.1656 - val_loss: 2.0309 - val_acc: 0.2574
Epoch 2/20
167/166 [==============================] - 20s 118ms/step - loss: 2.01
93 - acc: 0.2581 - val_loss: 1.8524 - val_acc: 0.3319
Epoch 3/20
167/166 [==============================] - 20s 118ms/step - loss: 1.83
39 - acc: 0.3292 - val_loss: 1.7042 - val_acc: 0.3758
Epoch 4/20
167/166 [==============================] - 20s 118ms/step - loss: 1.70
61 - acc: 0.3817 - val_loss: 1.5713 - val_acc: 0.4296
Epoch 5/20
167/166 [==============================] - 20s 121ms/step - loss: 1.58
88 - acc: 0.4222 - val_loss: 1.5889 - val_acc: 0.4208
Epoch 6/20
167/166 [==============================] - 20s 122ms/step - loss: 1.49
37 - acc: 0.4574 - val_loss: 1.5654 - val_acc: 0.4472
```

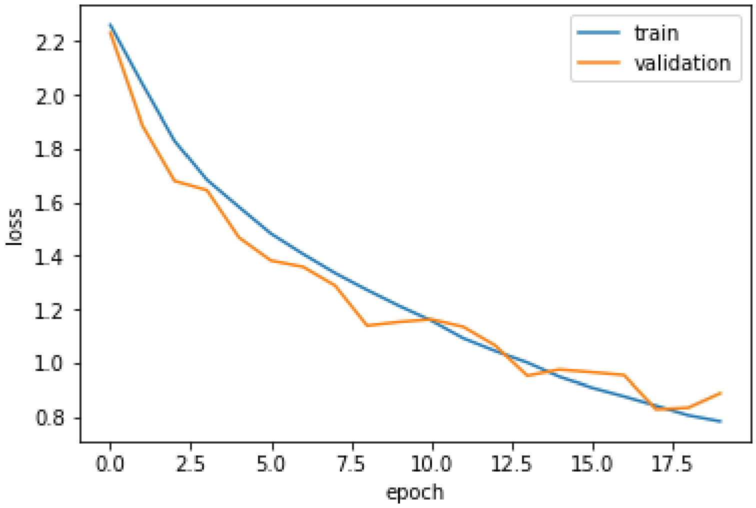
```
Epoch 7/20
167/166 [==============================] - 20s 121ms/step - loss: 1.41
83 - acc: 0.4897 - val_loss: 1.3250 - val_acc: 0.5220
Epoch 8/20
167/166 [==============================] - 20s 121ms/step - loss: 1.34
76 - acc: 0.5137 - val_loss: 1.3064 - val_acc: 0.5248
Epoch 9/20
167/166 [==============================] - 20s 120ms/step - loss: 1.28
42 - acc: 0.5389 - val_loss: 1.2504 - val_acc: 0.5591
Epoch 10/20
167/166 [==============================] - 20s 120ms/step - loss: 1.22
90 - acc: 0.5613 - val_loss: 1.1238 - val_acc: 0.5972
Epoch 11/20
167/166 [==============================] - 20s 120ms/step - loss: 1.16
44 - acc: 0.5846 - val_loss: 1.0949 - val_acc: 0.6155
Epoch 12/20
167/166 [==============================] - 20s 119ms/step - loss: 1.11
98 - acc: 0.6012 - val_loss: 1.0704 - val_acc: 0.6175
Epoch 13/20
167/166 [==============================] - 20s 120ms/step - loss: 1.07
01 - acc: 0.6185 - val_loss: 0.9877 - val_acc: 0.6496
Epoch 14/20
167/166 [==============================] - 20s 119ms/step - loss: 1.01
72 - acc: 0.6385 - val_loss: 0.9986 - val_acc: 0.6505
Epoch 15/20
167/166 [==============================] - 20s 120ms/step - loss: 0.97
50 - acc: 0.6549 - val_loss: 1.0942 - val_acc: 0.6233
Epoch 16/20
167/166 [==============================] - 20s 120ms/step - loss: 0.94
10 - acc: 0.6649 - val_loss: 0.9328 - val_acc: 0.6700
Epoch 17/20
167/166 [==============================] - 20s 120ms/step - loss: 0.90
16 - acc: 0.6799 - val_loss: 0.9447 - val_acc: 0.6669
Epoch 18/20
167/166 [==============================] - 20s 119ms/step - loss: 0.86
74 - acc: 0.6908 - val_loss: 1.0130 - val_acc: 0.6504
Epoch 19/20
167/166 [==============================] - 20s 119ms/step - loss: 0.83
77 - acc: 0.7030 - val_loss: 0.8350 - val_acc: 0.7118
Epoch 20/20
167/166 [==============================] - 20s 120ms/step - loss: 0.81
02 - acc: 0.7129 - val_loss: 0.8444 - val_acc: 0.7055
val loss: 0.844381419467926
val accuracy: 0.7055
```

In [17]:

```
import matplotlib.pyplot as plt
plt.plot(history2.history['acc'])
plt.plot(history2.history['val_acc'])
plt.ylabel('accuracy')
plt.xlabel('epoch')
plt.title('Neural network 2')
plt.legend(['train', 'validation'])
plt.show()

plt.plot(history2.history['loss'])
plt.plot(history2.history['val_loss'])
plt.ylabel('loss')
plt.xlabel('epoch')
```

```
plt.legend(['train', 'validation'])
plt.show()
```





Considering we went through 20 epochs now we see that we get diminishing returns that indicates an upper limit on accuracy. We also see that after 10 epochs we get poorer results than in (d) in problem 1. This indicates that the way we generate data is not doing so well. We do manage to get very close to the results in problem 1 though.

The accuracy increases to the very end.

In [72]:

```
predicted = model2.predict(x_val)
result = np.absolute(y_val-predicted)
errors = y_val[np.sum(result,1)>1.8]#1.8 er arbitrary tala milli 1 og 2
print(np.sum(errors,0))
```

```
[114.  85. 227. 193. 168. 163.  59.  36. 111.  31.]
```

Hér fyrir ofan erum við að skoða hvaða flokkar eiga það til að ruglast mest, sjáum að flokkur 3 er oftast misskilinn fyrir einhvern annan og flokkur nr. 10 er sjaldnast misskilinn.

3. [Learning to perform subtraction with an RNN, 30 points] Take a close look at the recurrent neural network code shown in class (https://keras.io/examples/addition_rnn/). It shows how sequence to sequence learning can be used to learn addition of small numbers. This is done by presenting the

network with input-output pairs, where the input is a string on the form "123+456" and the output is "579".

Modify the code for addition so that the network learns *subtraction* of three digit numbers. What is the validation accuracy after 30 epochs? You should mark all the modifications that you do to the code by inserting comments on the form

```
# START MOD
...
# END MOD
```

Comments:

- You need to modify the test that prevents the same example occurring multiple times in the list of "questions" (addition is commutative but subtraction is not).
- `val_acc` does not represent the fraction of correctly predicted validation examples (I have no idea why!) You need to write a short piece of code that sends all the examples in the validation set through the network and counts the number of mistakes.

I thought this problem was interesting and tried to improve the results by running 40 epochs.

In [13]:

```
from tensorflow.python.keras import layers
from tensorflow.python.keras.models import Sequential
import numpy as np


class CharacterTable(object):
    """Given a set of characters:
    + Encode them to a one-hot integer representation
    + Decode the one-hot or integer representation to their character output
    + Decode a vector of probabilities to their character output
    """
    def __init__(self, chars):
        """Initialize character table.

        # Arguments
            chars: Characters that can appear in the input.
        """
        self.chars = sorted(set(chars))
        self.char_indices = dict((c, i) for i, c in enumerate(self.chars))
        self.indices_char = dict((i, c) for i, c in enumerate(self.chars))

    def encode(self, C, num_rows):
        """One-hot encode given string C.

        # Arguments
            C: string, to be encoded.
            num_rows: Number of rows in the returned one-hot encoding. This is
                used to keep the # of rows for each data the same.
        """
        x = np.zeros((num_rows, len(self.chars)))
        for i, c in enumerate(C):
            x[i, self.char_indices[c]] = 1
        return x

    def decode(self, x, calc_argmax=True):
```

```
        """Decode the given vector or 2D array to their character output.

        # Arguments
            x: A vector or a 2D array of probabilities or one-hot representations;
                or a vector of character indices (used with `calc_argmax=False`).
            calc_argmax: Whether to find the character index with maximum
                probability, defaults to `True`.
        """
        if calc_argmax:
            x = x.argmax(axis=-1)
        return ''.join(self.indices_char[x] for x in x)


class colors:
    ok = '\033[92m'
    fail = '\033[91m'
    close = '\033[0m'

# Parameters for the model and dataset.
TRAINING_SIZE = 50000
DIGITS = 3
REVERSE = True

# Maximum length of input is 'int + int' (e.g., '345+678'). Maximum length of
# int is DIGITS.
MAXLEN = DIGITS + 1 + DIGITS

# All the numbers, plus sign and space for padding.
#START MOD
chars = '0123456789- '
#END MOD
ctable = CharacterTable(chars)

questions = []
expected = []
seen = set()
print('Generating data...')
while len(questions) < TRAINING_SIZE:
    f = lambda: int(''.join(np.random.choice(list('0123456789'))
                    for i in range(np.random.randint(1, DIGITS + 1))))
    a, b = f(), f()
    # Skip any addition questions we've already seen
    # Also skip any such that x+Y == Y+x (hence the sorting).
    #START MOD
    key = tuple((a, b))
    #END MOD
    if key in seen:
        continue
    seen.add(key)
    # Pad the data with spaces such that it is always MAXLEN.
    #START MOD
    q = '{}-{}'.format(a, b)
    query = q + ' ' * (MAXLEN - len(q))
    ans = str(a - b)
    #END MOD
    # Answers can be of maximum size DIGITS + 1.
    ans += ' ' * (DIGITS + 1 - len(ans))
    if REVERSE:
        # Reverse the query, e.g., '12+345  ' becomes '  543+21'. (Note the
        # space used for padding.)
        query = query[::-1]
```

```
    questions.append(query)
    expected.append(ans)
print('Total addition questions:', len(questions))

print('Vectorization...')
x = np.zeros((len(questions), MAXLEN, len(chars)), dtype=np.bool)
y = np.zeros((len(questions), DIGITS + 1, len(chars)), dtype=np.bool)
for i, sentence in enumerate(questions):
    x[i] = ctable.encode(sentence, MAXLEN)
for i, sentence in enumerate(expected):
    y[i] = ctable.encode(sentence, DIGITS + 1)

# Shuffle (x, y) in unison as the later parts of x will almost all be larger
# digits.
indices = np.arange(len(y))
np.random.shuffle(indices)
x = x[indices]
y = y[indices]

# Explicitly set apart 10% for validation data that we never train over.
split_at = len(x) - len(x) // 10
(x_train, x_val) = x[:split_at], x[split_at:]
(y_train, y_val) = y[:split_at], y[split_at:]

print('Training Data:')
print(x_train.shape)
print(y_train.shape)

print('Validation Data:')
print(x_val.shape)
print(y_val.shape)

# Try replacing GRU, or SimpleRNN.
RNN = layers.LSTM
HIDDEN_SIZE = 128
BATCH_SIZE = 128
LAYERS = 1

print('Build model...')
model = Sequential()
# "Encode" the input sequence using an RNN, producing an output of HIDDEN_SIZE.
# Note: In a situation where your input sequences have a variable length,
# use input_shape=(None, num_feature).
model.add(RNN(HIDDEN_SIZE, input_shape=(MAXLEN, len(chars))))
# As the decoder RNN's input, repeatedly provide with the last output of
# RNN for each time step. Repeat 'DIGITS + 1' times as that's the maximum
# length of output, e.g., when DIGITS=3, max output is 999+999=1998.
model.add(layers.RepeatVector(DIGITS + 1))
# The decoder RNN could be multiple layers stacked or a single layer.
for _ in range(LAYERS):
    # By setting return_sequences to True, return not only the last output but
    # all the outputs so far in the form of (num_samples, timesteps,
    # output_dim). This is necessary as TimeDistributed in the below expects
    # the first dimension to be the timesteps.
    model.add(RNN(HIDDEN_SIZE, return_sequences=True))

# Apply a dense layer to the every temporal slice of an input. For each of step
# of the output sequence, decide which character should be chosen.
model.add(layers.TimeDistributed(layers.Dense(len(chars), activation='softmax')))
model.compile(loss='categorical_crossentropy',
              optimizer='adam',
```

```
              metrics=['accuracy'])
model.summary()

# Train the model each generation and show predictions against the validation
# dataset.
for iteration in range(1, 40):
    print()
    print('-' * 50)
    print('Iteration', iteration)
    model.fit(x_train, y_train,
              batch_size=BATCH_SIZE,
              epochs=1,
              validation_data=(x_val, y_val))
    # Select 10 samples from the validation set at random so we can visualize
    # errors.
    for i in range(10):
        ind = np.random.randint(0, len(x_val))
        rowx, rowy = x_val[np.array([ind])], y_val[np.array([ind])]
        preds = model.predict_classes(rowx, verbose=0)
        q = ctable.decode(rowx[0])
        correct = ctable.decode(rowy[0])
        guess = ctable.decode(preds[0], calc_argmax=False)
        print('Q', q[::-1] if REVERSE else q, end=' ')
        print('T', correct, end=' ')
        if correct == guess:
            print(colors.ok + '☑' + colors.close, end=' ')
        else:
            print(colors.fail + '☒' + colors.close, end=' ')
        print(guess)

#Evaluate performance on the validation set
    corr=0
    for ind in range(len(x_val)):
        rowx, rowy = x_val[np.array([ind])], y_val[np.array([ind])]
        preds = model.predict_classes(rowx, verbose=0)
        q = ctable.decode(rowx[0])
        correct = ctable.decode(rowy[0])
        guess = ctable.decode(preds[0], calc_argmax=False)
        if correct == guess:
            corr+=1
    print("Validation accuracy:", corr/len(x_val))
```

```
Generating data...
Total addition questions: 50000
Vectorization...
Training Data:
(45000, 7, 12)
(45000, 4, 12)
Validation Data:
(5000, 7, 12)
(5000, 4, 12)
Build model...
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
lstm_2 (LSTM)                (None, 128)               72192
_________________________________________________________________
repeat_vector_1 (RepeatVecto (None, 4, 128)            0
_________________________________________________________________
lstm_3 (LSTM)                (None, 4, 128)            131584
_________________________________________________________________
time_distributed_1 (TimeDist (None, 4, 12)             1548
```

```
=================================================================
Total params: 205,324
Trainable params: 205,324
Non-trainable params: 0
_________________________________________________________________

--------------------------------------------------
Iteration 1
Train on 45000 samples, validate on 5000 samples
Epoch 1/1
45000/45000 [==============================] - 13s 297us/step - loss:
1.8684 - acc: 0.3458 - val_loss: 1.6285 - val_acc: 0.4037
Q 31-35   T -4   ☒ -1
Q 459-101 T 358  ☒ 118
Q 144-1   T 143  ☒ 188
Q 93-709  T -616 ☒ -278
Q 44-71   T -27  ☒ -1
Q 70-65   T 5    ☒ -1
Q 65-13   T 52   ☒ 11
Q 8-47    T -39  ☒ -1
Q 588-5   T 583  ☒ 888
Q 131-55  T 76   ☒ 11
Validation accuracy: 0.0032

--------------------------------------------------
Iteration 2
Train on 45000 samples, validate on 5000 samples
Epoch 1/1
45000/45000 [==============================] - 11s 254us/step - loss:
 1.5610 - acc: 0.4323 - val_loss: 1.4789 - val_acc: 0.4613
Q 64-633  T -569 ☒ -567
Q 83-771  T -688 ☒ -777
Q 302-96  T 206  ☒ 233
Q 301-608 T -307 ☒ -10
Q 421-21  T 400  ☒ 113
Q 10-206  T -196 ☒ -107
Q 44-774  T -730 ☒ -735
Q 357-696 T -339 ☒ -257
Q 296-97  T 199  ☒ 220
Q 14-788  T -774 ☒ -775
Validation accuracy: 0.01

--------------------------------------------------
Iteration 3
Train on 45000 samples, validate on 5000 samples
Epoch 1/1
45000/45000 [==============================] - 12s 261us/step - loss:
 1.4007 - acc: 0.4864 - val_loss: 1.3280 - val_acc: 0.5042
Q 87-17   T 70   ☑ 70
Q 251-812 T -561 ☒ -577
Q 33-952  T -919 ☒ -849
Q 456-897 T -441 ☒ -497
Q 10-266  T -256 ☒ -268
Q 893-57  T 836  ☒ 877
Q 872-6   T 866  ☒ 877
Q 9-743   T -734 ☒ -743
Q 0-543   T -543 ☒ -444
Q 15-474  T -459 ☒ -448
Validation accuracy: 0.017

--------------------------------------------------
```

```
Iteration 4
Train on 45000 samples, validate on 5000 samples
Epoch 1/1
45000/45000 [==============================] - 11s 255us/step - loss:
 1.2638 - acc: 0.5352 - val_loss: 1.2066 - val_acc: 0.5607
Q 9-350   T -341 ☒ -343
Q 9-350   T -341 ☒ -343
Q 7-49    T -42  ☒ -41
Q 428-934 T -506 ☒ -400
Q 8-180   T -172 ☒ -180
Q 571-47  T 524  ☒ 519
Q 78-170  T -92  ☒ -10
Q 39-603  T -564 ☒ -696
Q 582-65  T 517  ☒ 514
Q 462-812 T -350 ☒ -233
Validation accuracy: 0.0324

--------------------------------------------------
Iteration 5
Train on 45000 samples, validate on 5000 samples
Epoch 1/1
45000/45000 [==============================] - 11s 255us/step - loss:
 1.1622 - acc: 0.5753 - val_loss: 1.1243 - val_acc: 0.5895

Q 711-939 T -228 ☒ -133
Q 72-568  T -496 ☒ -403
Q 929-65  T 864  ☒ 877
Q 593-94  T 499  ☒ 401
Q 258-82  T 176  ☒ 211
Q 5-362   T -357 ☒ -351
Q 6-354   T -348 ☒ -341
Q 4-990   T -986 ☒ -990
Q 33-128  T -95  ☒ -11
Q 58-677  T -619 ☒ -610
Validation accuracy: 0.036

--------------------------------------------------
Iteration 6
Train on 45000 samples, validate on 5000 samples
Epoch 1/1
45000/45000 [==============================] - 12s 256us/step - loss:
 1.0690 - acc: 0.6149 - val_loss: 1.0262 - val_acc: 0.6319
Q 596-33  T 563  ☒ 569
Q 557-5   T 552  ☒ 554
Q 7-350   T -343 ☒ -349
Q 384-332 T 52   ☒ 11
Q 72-84   T -12  ☑ -12
Q 452-246 T 206  ☒ 271
Q 975-677 T 298  ☒ 190
Q 25-208  T -183 ☒ -197
Q 758-9   T 749  ☑ 749
Q 790-665 T 125  ☒ 11
Validation accuracy: 0.0484

--------------------------------------------------
Iteration 7
Train on 45000 samples, validate on 5000 samples
Epoch 1/1
45000/45000 [==============================] - 12s 256us/step - loss:
 0.9994 - acc: 0.6426 - val_loss: 0.9640 - val_acc: 0.6514
Q 50-833  T -783 ☒ -787
Q 0-522   T -522 ☒ -521
```

```
Q 547-0   T 547  ☒ 454
Q 62-644  T -582 ☒ -583
Q 31-808  T -777 ☒ -783
Q 19-93   T -74  ☒ -79
Q 16-16   T 0    ☒ -
Q 606-29  T 577  ☒ 589
Q 391-92  T 299  ☒ 298
Q 862-4   T 858  ☒ 856
Validation accuracy: 0.069

--------------------------------------------------
Iteration 8
Train on 45000 samples, validate on 5000 samples
Epoch 1/1
45000/45000 [==============================] - 12s 257us/step - loss:
 0.9365 - acc: 0.6683 - val_loss: 0.9089 - val_acc: 0.6768
Q 72-481  T -409 ☒ -413
Q 48-79   T -31  ☑ -31
Q 0-831   T -831 ☒ -838
Q 13-63   T -50  ☒ -53
Q 317-35  T 282  ☒ 280
Q 347-61  T 286  ☒ 288
Q 88-862  T -774 ☒ -772

Q 860-268 T 592  ☒ 683
Q 147-23  T 124  ☒ 136
Q 92-533  T -441 ☒ -456
Validation accuracy: 0.1

--------------------------------------------------
Iteration 9
Train on 45000 samples, validate on 5000 samples
Epoch 1/1
45000/45000 [==============================] - 12s 258us/step - loss:
 0.8745 - acc: 0.6902 - val_loss: 0.8401 - val_acc: 0.7027
Q 41-451  T -410 ☒ -413
Q 49-5    T 44   ☒ 43
Q 467-61  T 406  ☒ 403
Q 410-0   T 410  ☒ 400
Q 4-663   T -659 ☒ -658
Q 30-504  T -474 ☒ -471
Q 4-484   T -480 ☒ -489
Q 327-36  T 291  ☒ 290
Q 626-729 T -103 ☒ -11
Q 5-13    T -8   ☒ -1
Validation accuracy: 0.1372

--------------------------------------------------
Iteration 10
Train on 45000 samples, validate on 5000 samples
Epoch 1/1
45000/45000 [==============================] - 12s 261us/step - loss:
 0.8119 - acc: 0.7135 - val_loss: 0.7889 - val_acc: 0.7133
Q 30-87   T -57  ☑ -57
Q 21-338  T -317 ☒ -324
Q 0-162   T -162 ☑ -162
Q 556-24  T 532  ☒ 534
Q 722-979 T -257 ☒ -277
Q 76-890  T -814 ☒ -810
Q 56-445  T -389 ☒ -385
Q 628-88  T 540  ☒ 549
Q 79-50   T 29   ☒ 24
```

```
Q 768-43  T 725  ☑ 725
Validation accuracy: 0.162

--------------------------------------------------
Iteration 11
Train on 45000 samples, validate on 5000 samples
Epoch 1/1
45000/45000 [==============================] - 12s 258us/step - loss:
 0.7482 - acc: 0.7356 - val_loss: 0.7128 - val_acc: 0.7477
Q 4-892   T -888 ☑ -888
Q 160-473 T -313 ☒ -310
Q 470-319 T 151  ☒ 16
Q 343-7   T 336  ☑ 336
Q 56-445  T -389 ☒ -385
Q 66-205  T -139 ☒ -145
Q 61-9    T 52   ☒ 54
Q 408-463 T -55  ☒ -65
Q 8-305   T -297 ☒ -295
Q 9-800   T -791 ☒ -792
Validation accuracy: 0.2298

--------------------------------------------------
Iteration 12

Train on 45000 samples, validate on 5000 samples
Epoch 1/1
45000/45000 [==============================] - 12s 260us/step - loss:
 0.6820 - acc: 0.7581 - val_loss: 0.6453 - val_acc: 0.7716
Q 74-378  T -304 ☒ -303
Q 951-175 T 776  ☒ 775
Q 723-820 T -97  ☒ -1
Q 59-923  T -864 ☒ -865
Q 44-173  T -129 ☒ -116
Q 41-995  T -954 ☒ -955
Q 8-771   T -763 ☒ -764
Q 35-959  T -924 ☒ -926
Q 949-68  T 881  ☒ 889
Q 73-354  T -281 ☒ -288
Validation accuracy: 0.301

--------------------------------------------------
Iteration 13
Train on 45000 samples, validate on 5000 samples
Epoch 1/1
45000/45000 [==============================] - 12s 262us/step - loss:
 0.6009 - acc: 0.7903 - val_loss: 0.5725 - val_acc: 0.7980
Q 458-992 T -534 ☑ -534
Q 68-725  T -657 ☒ -655
Q 343-7   T 336  ☑ 336
Q 666-0   T 666  ☑ 666
Q 55-69   T -14  ☑ -14
Q 82-46   T 36   ☑ 36
Q 598-2   T 596  ☑ 596
Q 47-23   T 24   ☑ 24
Q 51-586  T -535 ☑ -535
Q 727-13  T 714  ☑ 714
Validation accuracy: 0.3752

--------------------------------------------------
Iteration 14
Train on 45000 samples, validate on 5000 samples
Epoch 1/1
```

```
45000/45000 [==============================] - 12s 262us/step - loss:
 0.5254 - acc: 0.8194 - val_loss: 0.5141 - val_acc: 0.8125
Q 598-2   T 596  ☑ 596
Q 7-921   T -914 ☒ -915
Q 581-329 T 252  ☒ 351
Q 894-4   T 890  ☑ 890
Q 479-20  T 459  ☒ 458
Q 6-161   T -155 ☑ -155
Q 880-133 T 747  ☒ 758
Q 290-304 T -14  ☒ -10
Q 47-308  T -261 ☒ -262
Q 64-55   T 9    ☒ 1
Validation accuracy: 0.4298

--------------------------------------------------
Iteration 15
Train on 45000 samples, validate on 5000 samples
Epoch 1/1
45000/45000 [==============================] - 12s 263us/step - loss:
 0.4506 - acc: 0.8481 - val_loss: 0.4244 - val_acc: 0.8560
Q 2-76    T -74  ☑ -74
Q 70-871  T -801 ☒ -800
Q 556-748 T -192 ☒ -191

Q 38-143  T -105 ☒ -106
Q 95-597  T -502 ☒ -503
Q 880-41  T 839  ☑ 839
Q 96-315  T -219 ☑ -219
Q 882-497 T 385  ☒ 384
Q 506-73  T 433  ☑ 433
Q 157-69  T 88   ☒ 78
Validation accuracy: 0.5524

--------------------------------------------------
Iteration 16
Train on 45000 samples, validate on 5000 samples
Epoch 1/1
45000/45000 [==============================] - 12s 261us/step - loss:
 0.3810 - acc: 0.8747 - val_loss: 0.3493 - val_acc: 0.8870
Q 329-91  T 238  ☒ 228
Q 23-430  T -407 ☑ -407
Q 68-614  T -546 ☑ -546
Q 583-28  T 555  ☑ 555
Q 567-9   T 558  ☑ 558
Q 92-724  T -632 ☑ -632
Q 32-273  T -241 ☑ -241
Q 485-39  T 446  ☑ 446
Q 16-47   T -31  ☑ -31
Q 631-52  T 579  ☒ 589
Validation accuracy: 0.6464

--------------------------------------------------
Iteration 17
Train on 45000 samples, validate on 5000 samples
Epoch 1/1
45000/45000 [==============================] - 12s 265us/step - loss:
 0.3148 - acc: 0.9006 - val_loss: 0.2851 - val_acc: 0.9129
Q 414-18  T 396  ☑ 396
Q 81-13   T 68   ☑ 68
Q 529-64  T 465  ☑ 465
Q 26-370  T -344 ☑ -344
Q 424-624 T -200 ☑ -200
```

```
Q 453-27   T 426  ☑ 426
Q 30-550   T -520 ☑ -520
Q 486-5    T 481  ☑ 481
Q 385-76   T 309  ☑ 309
Q 274-44   T 230  ☑ 230
Validation accuracy: 0.7274

--------------------------------------------------
Iteration 18
Train on 45000 samples, validate on 5000 samples
Epoch 1/1
45000/45000 [==============================] - 12s 269us/step - loss:
 0.2598 - acc: 0.9220 - val_loss: 0.2504 - val_acc: 0.9222
Q 329-35   T 294  ☑ 294
Q 675-257 T 418  ☑ 418
Q 496-7    T 489  ☑ 489
Q 2-282    T -280 ☑ -280
Q 90-202   T -112 ☒ -113
Q 287-66   T 221  ☑ 221
Q 3-57     T -54  ☑ -54
Q 31-808   T -777 ☑ -777
Q 35-860   T -825 ☒ -824
Q 533-732 T -199 ☒ -299

Validation accuracy: 0.7512

--------------------------------------------------
Iteration 19
Train on 45000 samples, validate on 5000 samples
Epoch 1/1
45000/45000 [==============================] - 12s 276us/step - loss:
 0.2137 - acc: 0.9402 - val_loss: 0.2012 - val_acc: 0.9419
Q 678-8    T 670  ☑ 670
Q 350-170 T 180  ☑ 180
Q 214-6    T 208  ☑ 208
Q 60-96    T -36  ☑ -36
Q 433-73   T 360  ☑ 360
Q 781-34   T 747  ☑ 747
Q 37-381   T -344 ☑ -344
Q 562-9    T 553  ☑ 553
Q 742-26   T 716  ☑ 716
Q 83-157   T -74  ☑ -74
Validation accuracy: 0.8098

--------------------------------------------------
Iteration 20
Train on 45000 samples, validate on 5000 samples
Epoch 1/1
45000/45000 [==============================] - 12s 271us/step - loss:
 0.1746 - acc: 0.9541 - val_loss: 0.1675 - val_acc: 0.9547
Q 199-57   T 142  ☑ 142
Q 242-558 T -316 ☑ -316
Q 836-418 T 418  ☒ 408
Q 385-47   T 338  ☑ 338
Q 99-917   T -818 ☑ -818
Q 229-1    T 228  ☑ 228
Q 325-6    T 319  ☑ 319
Q 765-56   T 709  ☑ 709
Q 287-536 T -249 ☒ -248
Q 806-8    T 798  ☑ 798
Validation accuracy: 0.847
```

```
--------------------------------------------------
Iteration 21
Train on 45000 samples, validate on 5000 samples
Epoch 1/1
45000/45000 [==============================] - 12s 269us/step - loss:
 0.1481 - acc: 0.9628 - val_loss: 0.1454 - val_acc: 0.9632
Q 41-26   T 15   ☑ 15
Q 845-371 T 474  ☑ 474
Q 6-59    T -53  ☑ -53
Q 520-26  T 494  ☑ 494
Q 229-16  T 213  ☑ 213
Q 131-18  T 113  ☑ 113
Q 89-1    T 88   ☑ 88
Q 959-335 T 624  ☒ 634
Q 2-139   T -137 ☑ -137
Q 635-64  T 571  ☑ 571
Validation accuracy: 0.8788

--------------------------------------------------
Iteration 22
Train on 45000 samples, validate on 5000 samples
Epoch 1/1
45000/45000 [==============================] - 12s 269us/step - loss:

 0.1249 - acc: 0.9697 - val_loss: 0.1251 - val_acc: 0.9667
Q 4-81    T -77  ☑ -77
Q 22-34   T -12  ☑ -12
Q 79-575  T -496 ☑ -496
Q 285-782 T -497 ☑ -497
Q 6-728   T -722 ☑ -722
Q 99-130  T -31  ☑ -31
Q 45-516  T -471 ☑ -471
Q 96-64   T 32   ☑ 32
Q 964-70  T 894  ☑ 894
Q 834-982 T -148 ☑ -148
Validation accuracy: 0.886

--------------------------------------------------
Iteration 23
Train on 45000 samples, validate on 5000 samples
Epoch 1/1
45000/45000 [==============================] - 12s 269us/step - loss:
 0.1104 - acc: 0.9731 - val_loss: 0.1038 - val_acc: 0.9741
Q 71-49   T 22   ☑ 22
Q 385-200 T 185  ☑ 185
Q 350-457 T -107 ☑ -107
Q 43-115  T -72  ☑ -72
Q 55-132  T -77  ☑ -77
Q 803-62  T 741  ☑ 741
Q 323-52  T 271  ☑ 271
Q 0-944   T -944 ☑ -944
Q 71-573  T -502 ☑ -502
Q 27-160  T -133 ☒ -134
Validation accuracy: 0.91

--------------------------------------------------
Iteration 24
Train on 45000 samples, validate on 5000 samples
Epoch 1/1
45000/45000 [==============================] - 12s 272us/step - loss:
 0.0977 - acc: 0.9759 - val_loss: 0.1012 - val_acc: 0.9717
Q 15-389  T -374 ☑ -374
```

```
Q 12-67   T -55  ☑ -55
Q 2-514   T -512 ☑ -512
Q 48-696  T -648 ☑ -648
Q 496-65  T 431  ☑ 431
Q 374-65  T 309  ☑ 309
Q 32-93   T -61  ☑ -61
Q 112-3   T 109  ☑ 109
Q 31-35   T -4   ☑ -4
Q 2-439   T -437 ☑ -437
Validation accuracy: 0.902

--------------------------------------------------
Iteration 25
Train on 45000 samples, validate on 5000 samples
Epoch 1/1
45000/45000 [==============================] - 12s 269us/step - loss:
 0.0815 - acc: 0.9812 - val_loss: 0.1054 - val_acc: 0.9684
Q 595-626 T -31  ☑ -31
Q 559-68  T 491  ☑ 491
Q 36-938  T -902 ☑ -902
Q 83-771  T -688 ☑ -688
Q 834-982 T -148 ☑ -148
Q 768-43  T 725  ☑ 725

Q 64-766  T -702 ☑ -702
Q 9-203   T -194 ☑ -194
Q 481-21  T 460  ☑ 460
Q 92-935  T -843 ☑ -843
Validation accuracy: 0.8874

--------------------------------------------------
Iteration 26
Train on 45000 samples, validate on 5000 samples
Epoch 1/1
45000/45000 [==============================] - 12s 270us/step - loss:
 0.0726 - acc: 0.9836 - val_loss: 0.0715 - val_acc: 0.9818
Q 505-87  T 418  ☒ 428
Q 1-261   T -260 ☑ -260
Q 82-46   T 36   ☑ 36
Q 845-371 T 474  ☑ 474
Q 869-32  T 837  ☑ 837
Q 27-96   T -69  ☑ -69
Q 373-70  T 303  ☑ 303
Q 5-986   T -981 ☑ -981
Q 120-829 T -709 ☑ -709
Q 6-310   T -304 ☑ -304
Validation accuracy: 0.9344

--------------------------------------------------
Iteration 27
Train on 45000 samples, validate on 5000 samples
Epoch 1/1
45000/45000 [==============================] - 13s 281us/step - loss:
 0.0704 - acc: 0.9832 - val_loss: 0.1437 - val_acc: 0.9508
Q 338-332 T 6    ☒ 96
Q 19-261  T -242 ☑ -242
Q 993-39  T 954  ☑ 954
Q 367-395 T -28  ☑ -28
Q 629-360 T 269  ☒ 279
Q 8-425   T -417 ☒ -416
Q 23-461  T -438 ☑ -438
Q 96-68   T 28   ☑ 28
```

```
Q 8-722   T -714 ☑ -714
Q 496-65  T 431  ☑ 431
Validation accuracy: 0.8314

--------------------------------------------------
Iteration 28
Train on 45000 samples, validate on 5000 samples
Epoch 1/1
45000/45000 [==============================] - 12s 276us/step - loss:
 0.0720 - acc: 0.9816 - val_loss: 0.0689 - val_acc: 0.9804
Q 14-240  T -226 ☑ -226
Q 506-799 T -293 ☑ -293
Q 7-108   T -101 ☑ -101
Q 83-8    T 75   ☑ 75
Q 479-61  T 418  ☑ 418
Q 64-83   T -19  ☑ -19
Q 808-667 T 141  ☑ 141
Q 780-87  T 693  ☑ 693
Q 754-986 T -232 ☑ -232
Q 11-539  T -528 ☑ -528
Validation accuracy: 0.9316

--------------------------------------------------

Iteration 29
Train on 45000 samples, validate on 5000 samples
Epoch 1/1
45000/45000 [==============================] - 12s 269us/step - loss:
 0.0454 - acc: 0.9914 - val_loss: 0.0650 - val_acc: 0.9827
Q 24-556  T -532 ☑ -532
Q 97-64   T 33   ☑ 33
Q 866-433 T 433  ☑ 433
Q 433-51  T 382  ☑ 382
Q 474-8   T 466  ☑ 466
Q 73-921  T -848 ☑ -848
Q 10-143  T -133 ☑ -133
Q 264-78  T 186  ☑ 186
Q 964-508 T 456  ☑ 456
Q 62-36   T 26   ☑ 26
Validation accuracy: 0.939

--------------------------------------------------
Iteration 30
Train on 45000 samples, validate on 5000 samples
Epoch 1/1
45000/45000 [==============================] - 12s 274us/step - loss:
 0.0551 - acc: 0.9862 - val_loss: 0.0736 - val_acc: 0.9766
Q 901-692 T 209  ☒ 219
Q 658-38  T 620  ☑ 620
Q 970-674 T 296  ☑ 296
Q 627-7   T 620  ☑ 620
Q 436-6   T 430  ☑ 430
Q 4-880   T -876 ☑ -876
Q 170-58  T 112  ☑ 112
Q 462-91  T 371  ☑ 371
Q 31-676  T -645 ☑ -645
Q 244-9   T 235  ☑ 235
Validation accuracy: 0.9138

--------------------------------------------------
Iteration 31
Train on 45000 samples, validate on 5000 samples
```

```
Epoch 1/1
45000/45000 [==============================] - 12s 272us/step - loss:
 0.0611 - acc: 0.9838 - val_loss: 0.0621 - val_acc: 0.9821
Q 349-899 T -550 ☒ -540
Q 5-17    T -12  ☑ -12
Q 49-460  T -411 ☑ -411
Q 5-429   T -424 ☑ -424
Q 357-696 T -339 ☑ -339
Q 321-55  T 266  ☑ 266
Q 0-335   T -335 ☑ -335
Q 8-254   T -246 ☑ -246
Q 41-29   T 12   ☑ 12
Q 79-237  T -158 ☑ -158
Validation accuracy: 0.939

--------------------------------------------------
Iteration 32
Train on 45000 samples, validate on 5000 samples
Epoch 1/1
45000/45000 [==============================] - 13s 279us/step - loss:
 0.0345 - acc: 0.9938 - val_loss: 0.0401 - val_acc: 0.9903
Q 570-72  T 498  ☑ 498
Q 640-4   T 636  ☑ 636

Q 9-74    T -65  ☑ -65
Q 857-505 T 352  ☑ 352
Q 2-761   T -759 ☑ -759
Q 69-16   T 53   ☑ 53
Q 406-804 T -398 ☑ -398
Q 6-632   T -626 ☑ -626
Q 541-158 T 383  ☑ 383
Q 572-3   T 569  ☑ 569
Validation accuracy: 0.9644

--------------------------------------------------
Iteration 33
Train on 45000 samples, validate on 5000 samples
Epoch 1/1
45000/45000 [==============================] - 13s 279us/step - loss:
 0.0296 - acc: 0.9950 - val_loss: 0.0416 - val_acc: 0.9892
Q 96-489  T -393 ☑ -393
Q 410-0   T 410  ☑ 410
Q 444-52  T 392  ☑ 392
Q 164-41  T 123  ☑ 123
Q 97-340  T -243 ☑ -243
Q 294-68  T 226  ☑ 226
Q 9-496   T -487 ☑ -487
Q 61-570  T -509 ☑ -509
Q 548-58  T 490  ☑ 490
Q 9-486   T -477 ☑ -477
Validation accuracy: 0.96

--------------------------------------------------
Iteration 34
Train on 45000 samples, validate on 5000 samples
Epoch 1/1
45000/45000 [==============================] - 13s 292us/step - loss:
 0.0407 - acc: 0.9906 - val_loss: 0.0468 - val_acc: 0.9860
Q 34-309  T -275 ☑ -275
Q 572-34  T 538  ☑ 538
Q 19-8    T 11   ☑ 11
Q 89-573  T -484 ☑ -484
```

```
Q 905-6   T 899  ☑ 899
Q 73-32   T 41   ☑ 41
Q 40-59   T -19  ☑ -19
Q 3-249   T -246 ☒ -245
Q 392-6   T 386  ☑ 386
Q 48-3    T 45   ☑ 45
Validation accuracy: 0.9482

--------------------------------------------------
Iteration 35
Train on 45000 samples, validate on 5000 samples
Epoch 1/1
45000/45000 [==============================] - 13s 282us/step - loss:
 0.0327 - acc: 0.9928 - val_loss: 0.0571 - val_acc: 0.9820
Q 749-7   T 742  ☑ 742
Q 572-3   T 569  ☑ 569
Q 80-828  T -748 ☑ -748
Q 84-83   T 1    ☑ 1
Q 726-3   T 723  ☑ 723
Q 796-206 T 590  ☑ 590
Q 433-9   T 424  ☑ 424
Q 930-6   T 924  ☑ 924
Q 9-496   T -487 ☑ -487

Q 4-157   T -153 ☑ -153
Validation accuracy: 0.933

--------------------------------------------------
Iteration 36
Train on 45000 samples, validate on 5000 samples
Epoch 1/1
45000/45000 [==============================] - 13s 283us/step - loss:
 0.0320 - acc: 0.9931 - val_loss: 0.0371 - val_acc: 0.9895
Q 3-908   T -905 ☑ -905
Q 962-67  T 895  ☑ 895
Q 584-50  T 534  ☑ 534
Q 196-943 T -747 ☒ -746
Q 494-70  T 424  ☑ 424
Q 555-163 T 392  ☑ 392
Q 35-689  T -654 ☑ -654
Q 333-48  T 285  ☑ 285
Q 842-991 T -149 ☑ -149
Q 80-687  T -607 ☑ -607
Validation accuracy: 0.962

--------------------------------------------------
Iteration 37
Train on 45000 samples, validate on 5000 samples
Epoch 1/1
45000/45000 [==============================] - 13s 285us/step - loss:
 0.0452 - acc: 0.9881 - val_loss: 0.0280 - val_acc: 0.9933
Q 278-20  T 258  ☑ 258
Q 9-309   T -300 ☑ -300
Q 8-205   T -197 ☒ -187
Q 740-49  T 691  ☑ 691
Q 41-451  T -410 ☑ -410
Q 151-425 T -274 ☑ -274
Q 641-51  T 590  ☑ 590
Q 669-88  T 581  ☑ 581
Q 632-7   T 625  ☑ 625
Q 88-2    T 86   ☑ 86
Validation accuracy: 0.9752
```

```
--------------------------------------------------
Iteration 38
Train on 45000 samples, validate on 5000 samples
Epoch 1/1
45000/45000 [==============================] - 13s 284us/step - loss:
 0.0197 - acc: 0.9971 - val_loss: 0.0240 - val_acc: 0.9940
Q 802-401 T 401  ☑ 401
Q 4-990   T -986 ☑ -986
Q 579-68  T 511  ☑ 511
Q 410-42  T 368  ☑ 368
Q 49-319  T -270 ☑ -270
Q 5-922   T -917 ☑ -917
Q 82-684  T -602 ☑ -602
Q 6-298   T -292 ☑ -292
Q 3-501   T -498 ☑ -498
Q 163-6   T 157  ☑ 157
Validation accuracy: 0.9772

--------------------------------------------------
Iteration 39
Train on 45000 samples, validate on 5000 samples
Epoch 1/1
45000/45000 [==============================] - 12s 277us/step - loss:
 0.0163 - acc: 0.9979 - val_loss: 0.0222 - val_acc: 0.9950
Q 41-64   T -23  ☑ -23
Q 9-74    T -65  ☑ -65
Q 32-906  T -874 ☑ -874
Q 71-72   T -1   ☑ -1
Q 903-855 T 48   ☒ 58
Q 24-609  T -585 ☑ -585
Q 11-40   T -29  ☑ -29
Q 494-70  T 424  ☑ 424
Q 88-45   T 43   ☑ 43
Q 484-939 T -455 ☑ -455
Validation accuracy: 0.9806
```

After 30 Iteration we have accuracy 0.9138 which is pretty good considering we started in 0.0032, but after the 30th epoch we steadily rise to 0.9806 in validation accuracy when we hit iteration number 39. It's likely that you can get as accurate as you want if you give yourself sufficient amount of iterations.