Intro to Deep Learning Assignment 2

Student Number:13064947, email:john.goodacre.13@ucl.ac.uk

December 2017

1 Training/Optimization: Qu 1

Recall the simple neural network from Assignment 1 Question 6 that used the MNIST dataset and the following architecture:

- Input layer: 784 values
- Hidden layer 1: 100 fully-connected neurons, with sigmoid activation
- Hidden layer 2: 50 fully-connected neurons, with ReLU activation
- Output layer: 10 fully-connected neurons, with softmax activation

1.1 Calculate cross-entropy loss

With N training examples and K classes the cross-entropy loss is as follows:

$$L(X,Y) = \left(-\frac{1}{N}\right)\sum_{i=1}^{N}\sum_{k=1}^{K} y^{(k)}_{(i,true)} log(y^{(k)}_{(i,predict)})$$

Or in code, I will simply be minimising the following function:

```
def cost(y_truth, y_pred):
    return -((y_truth * np.log(y_pred)).sum())/y_pred.shape[0]
```

1.2 Use the backpropagation algorithm to calculate the gradients of this loss with respect to the network weights and update the weights

1.2.1 Forward Prop

To do backprop we need to do a forward propogation through the network first. Here we pass weights, biases and activation functions and sweep forwards through the network starting from the data X. a[-1] will of course be our final predictions.

```
def forward(X, weights, biases, act):
    a = [X]
    for w,b,f in zip(weights, biases, act):
        a.append(f[0](a[-1].dot(w) + b))
    return a
```

1.2.2 Gradients

The activation functions and their respective gradients are coded as given below, note that in my backprop implementation I treated the outer layer as a separate case thus didn't code an explicit softmax prime.

```
def sigmoid(x): return 1/(1 + np.exp(-x))
def sigmoid_prime(x): return sigmoid(x) * (1 - sigmoid(x))

def relu(x): return np.maximum(x, 0)
def relu_prime(x): return np.greater(x, 0).astype(int)

def softmax(x):
    expx = np.exp(x)
    return expx / expx.sum(axis=1, keepdims=True)
```

1.3 Randomly initialise the weights

Weights, biases and activation functions (and their derivatives) are initialised and passed as given below. Note that the network can be sensitive to initialisation, small random numbers close to zero being more stable.

```
weights = [np.random.randn(*w) * 0.1 for w in [(784, 100),
             (100, 50), (50,10)]]
biases = [np.random.randn(*b) * 0.1 for b in [(100,),
             (50,), (10,)]]
activations = [(sigmoid, sigmoid_prime), (relu, relu_prime),
             (softmax, None)]
```

1.4 Train the network architecture described above on the MNIST dataset for several epochs. (The number of epochs is to be determined by an easily altered parameter in your code.)

1.4.1 Backprop

After initialisation the bulk of the work is done in the gradients function. Here training examples X, and targets Y are passed together with the randomly initialises weights, biases and the derivatives of the activation functions. We sweep backwards through the network using the chain rule to recursively update the gradients of the weights and biases of the network.

```
def gradients(X, Y, weights, biases, act):
    #forward
    a = forward(X, weights, biases, activations)

    #backprop
    gradients = np.empty_like(weights)
    grad_bias = np.empty_like(biases)

    #output layer
    delta = a[-1] - Y
    gradients[-1] = a[-2].T.dot(delta)
    grad_bias[-1] = np.sum(delta, axis=0)

    for i in range(len(a)-2,0,-1):
        delta = act[i-1][1](a[i]) * delta.dot(weights[i].T)
        grad_bias[i-1] = np.sum(delta, axis=0)
        gradients[i-1] = a[i-1].T.dot(delta)

    grad_bias = grad_bias / len(X)
    gradients = gradients / len(X)

    return gradients, grad_bias
```

This function is used in the for loop given below. We choose our number of epochs, our batch size and construct our training data. The gradients of the weights and biases are gathered by the functions already described above. The weights and biases are then adjusted via gradient descent (using a pre-specified learning rate). Again the network will be sensitive to the learning rate. Note that it would be quicker if I also implemented momentum, but this was not asked for in this question.

```
for i in range(num_epochs):
    for j in range(0, len(trainX), batch_size):
        X, Y = trainX[j:j+batch_size], trainY[j:j+batch_size]

        gradw, grad_bias = gradients(X, Y, weights, biases,
                                     activations)
        biases -= learn_rate * grad_bias
        weights -= learn_rate * gradw
```

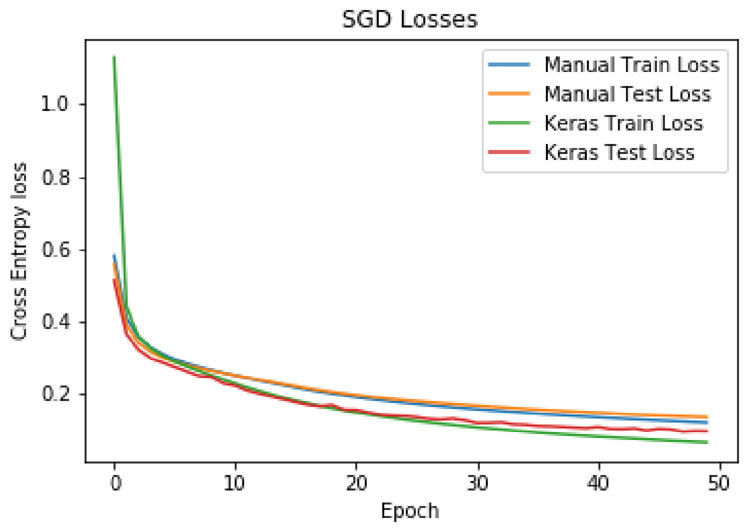


Figure 1: Stochastic Gradient Descent Losses

1.5 Include a figure that is produced by executing this code that shows the change in loss at each epoch during training. Compare in this figure to the same per-epoch evolution of loss under an optimizer built into Keras, briefly comment

Fig 1, shows the cross-entropy loss per epoch of training. I ran a simple stochastic (batch) gradient descent without momentum. First, for the manual MLP implemented above, then for a matching Keras implementation. The learning rates, batch sizes and number of epochs were set to the same numbers for the manual implementation and for Keras (0.05, 100, 50 respectively)

The training and test cross-entropy losses are shown for the manual and Keras implementation. The Keras implementation was deliberately 'disabled' to match the manual one. For example, I removed momentum and other features that would have sped up the optimisation.

Fig 2, shows the accuracies per epoch, for the training and test data for the same experiment.

For interests sake Fig 3, again shows the losses per epoch. However, in this example I changed the Keras optimiser to RMSprop(). The figure shows that given this different optimiser and momentum that the optimiser learns far quicker.

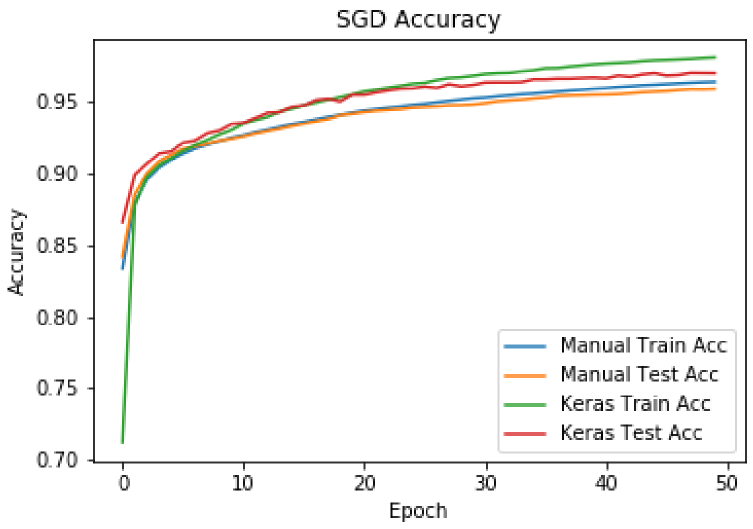


Figure 2: Stochastic Gradient Descent Accuracies

One caveat. In this case we can see that 50 epochs are causing over-fitting. The model is learning the training data, but its ability to generalise on the test set is deteriorating after a few epochs. This could of course be mitigated by using dropout, which I didn't do as I wanted to compare identical models as per the question.

Fig 4, shows the accuracies for the RMSProp optimiser. As expected because of the faster learning but eventual over-fitting the accuracy on the test set drops off after several epochs.

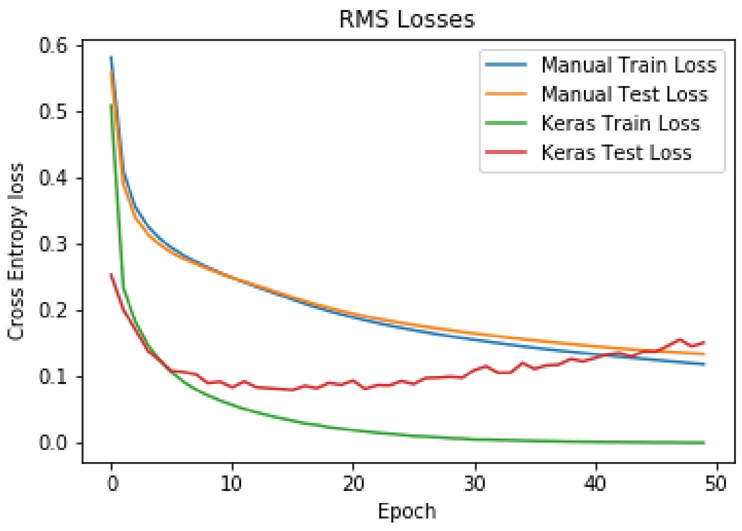


Figure 3: RMS Losses

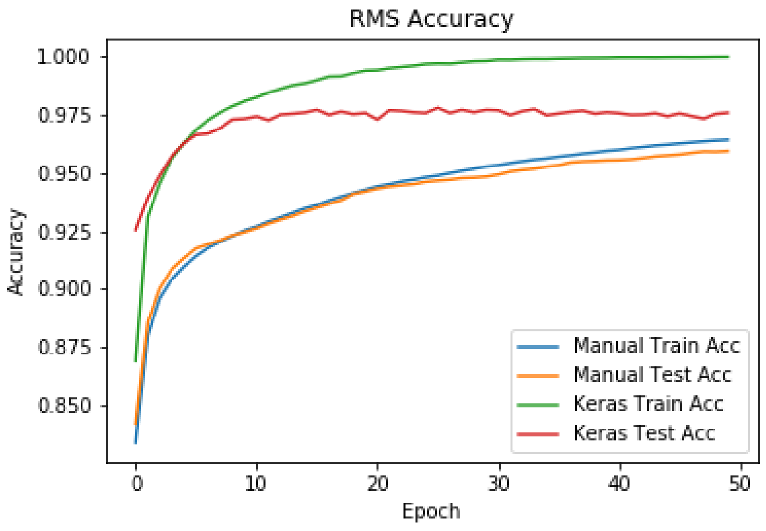


Figure 4: RMS Accuracies

1.5.1 Keras Implementation

For completeness I give the code for the Keras implementation below.

```
model = Sequential()
model.add(Dense(100, activation='sigmoid', input_shape=(784,)))
model.add(Dense(50, activation='relu'))
model.add(Dense(num_classes, activation='softmax'))

model.summary()

model.compile(loss='categorical_crossentropy',
              optimizer=RMSprop(),
              metrics=['accuracy'])

history = model.fit(trainX, trainY,
                    batch_size=batch_size,
                    epochs=num_epochs,
                    verbose=1,
                    validation_data=(testX, testY))
```

1.5.2 Comments

Clearly the implementation is fairly basic. I have not included momentum in the optimisation, only basic gradient descent. As I made the Keras optimiser more powerful we saw the model learned quickly but over-fitted, and should really include dropout layers. It is hard to make brief comments as optimisation is a huge subject. We are finding a local minima in a non-convex error surface.

There are various tricks/ heuristics to this depending upon the structure of the problem. Momentum, batch normalisation, gradient clipping, decay of the learning rate, batch sizes etc. etc. Clearly here we have only set up the most basic implementation and any of these tweaks would result in different losses per epoch.

2 Improving Network Accuracy Qu 2

Start by cloning the following repository: https://github.com/PetarV-/L42-Starter-Pack. Follow the instructions in the README of this repository to setup and execute the model.

Task. Modify the model architecture to exceed 92 per cent accuracy on the validation set by applying some of the techniques described in Lecture 6 to parts of the architecture (for example, dropout and batch normalisation to name just two.)
Provide: (i) a description of techniques adopted that enabled you to exceed the 92 per cent level; (ii) precisely how they are applied within the architecture; and (iii), justify the design decisions you made.

2.1 Approach

I noticed that the NotMNIST training set given had only 13,000 training examples, also that the data set comprises images. Thus my basic approach was to build the largest model possible without over-fitting the data, because the data comprised images I chose to use a convolutional neural net with its obvious advantages and not least stellar results on the easier MNIST dataset. Anecdotally 92% on the validation data is pretty easy, even a large multi-layer perceptron can do it, getting over 95% took more time and was trickier.

2.2 Model

The model itself was a bit of a beast. First I will present the convolutional part of the model. In the segment of code given below I used three convolutional layers, with two layers of max-pooling. I chose kernel sizes of 4x4 and feature maps of 32, 64, 128 respectively. The basic motivation for the model was known models for vision such as Le-Net5 amongst many.

2.3 Convolution part

```
conv_1 = Conv2D(conv_depth1, (kernel_size, kernel_size),
                               padding='same',
                               kernel_initializer='he_uniform',
                               kernel_regularizer=l2(l2_lambda),
                               activation='relu')(inp_bnorm)
        conv_1 = BatchNormalization()(conv_1)
        conv_2 = Conv2D(conv_depth2, (kernel_size, kernel_size),
                               padding='same',
                               kernel_initializer='he_uniform',
                               kernel_regularizer=l2(l2_lambda),
                               activation='relu')(conv_1)
        conv_2 = BatchNormalization()(conv_2)
        pool_1 = MaxPooling2D(pool_size=(pool_size, pool_size))(
                                          conv_2)
        drop_1 = Dropout(drop_prob_1)(pool_1)
        conv_3 = Conv2D(conv_depth3, (kernel_size, kernel_size),
                        padding='same',
                        kernel_initializer='he_uniform',
                        kernel_regularizer=l2(l2_lambda),
                        activation='relu')(drop_1)

        conv_3 = BatchNormalization()(conv_3)
        pool_2 = MaxPooling2D(pool_size=(pool_size, pool_size))(
                                          conv_3)
        drop_2 = Dropout(drop_prob_1)(pool_2)
```

2.3.1 Activation function

You may also notice that I chose relu activation functions and used dropout and batch normalisation. For the convolutional part of the neural net I used quite a low dropout of 0.2, low relative to fully connected layers at least. Here I am basically following people like Hinton and in this case it seemed to work well. Relu tends to be my 'go to' activation and seems to work well generally. Out of curiosity I tried more advanced activations such as PRelu() and Elu(), but I saw little difference in results. Batch normalisation clearly helped the neural net and improved results.

2.3.2 Regularisation

As regards dropout I also tried new implementations such as Yarin Gal's concrete dropout [2], but again this gave no improvement by the time I had over a 95% accuracy. I felt that dropout was necessary in general with a large model as otherwise I could quickly 'remember' the training data to 100% accuracy but have poor generalisation.

Although dropout is a form of regularisation in itself as regards regularisation I also added in a standard L2 regularisation. At this point though we are really talking quite minute percentage gains and I consider this a nicety rather than a necessity for the model.

2.4 Fully Connected Layers

The output from the convolutional net was flatten and put into a two hidden layer fully connected network, with 512 and 256 neurons respectively. Of course this was then pushed out into a softmax layer for classification. I used higher levels of dropout for the fully connected layers, settling at 0.4.

2.4.1 Initialisation

In terms of initialisation of the network, I chose samples from the truncated normal and uniform distributions, mostly motivated by the lecture on being nice to your neurons!

```
flat = Flatten()(drop_2)
        hidden = Dense(hidden_size1,
                       kernel_initializer='he_uniform',
                       kernel_regularizer=l2(l2_lambda),
                       activation='relu')(flat)
        hidden = BatchNormalization()(hidden)
        drop_2 = Dropout(drop_prob_2)(hidden)
        hidden1 = Dense(hidden_size2,
                       kernel_initializer='he_uniform',
                       kernel_regularizer=l2(l2_lambda),
                       activation='relu')(drop_2)
        hidden1 = BatchNormalization()(hidden1)
        drop3 = Dropout(drop_prob_2)(hidden1)


        outputs.append(Dense(nb_classes,
                          kernel_initializer='glorot_uniform',
                          kernel_regularizer=l2(l2_lambda),
                          activation='softmax')(drop3))
```

2.4.2 Ensembling

If one looks closely then they may notice that I have a list of outputs, not one. I decided that three heads are better than one and used an emsemble of three of these rather large nets.

2.4.3 Data Augmentation

Because we have a limited amount of training examples, I made use of Keras data augmentation abilities. The code below augments the training data by shifting and rotating images, whilst ideally maintaining the same label.

```
datagen = ImageDataGenerator(width_shift_range=0.1,
                height_shift_range=0.1, rotation_range=10)
datagen.fit(X_train)

# Load and compile model
model = get_model()
```

2.4.4 Optimisation

For optimisation I have previously used Adam which works well, however Nadam was introduced in the lecture (Adam with nesterov momentum), so I chose to use this - I briefly tried other optimisers like Adam and RMPProp but didn't feel the need to do much here as the model already had good performance. The code below illustrates the optimiser choice as well as the use of the datagen facility. I used a batch size of 128 and ran the model for 100 epochs. I put an early stopping callback on the model with a high degree of patience so that I didn't have to sit there all day watching it!

```
model.compile(loss='categorical_crossentropy',
              optimizer=Nadam(),
              metrics=['accuracy'])


model.fit_generator(datagen.flow(X_train, y_train, batch_size=
                                  batch_size),steps_per_epoch=len(
                                  X_train) / batch_size,
                    epochs=nb_epoch, verbose=1,
                    validation_data=(X_test, y_test),
                    callbacks=[EarlyStopping(monitor='val_loss',
                                                   patience=20)])
```

In summary, I was pretty sure that a large conv net with Relu activations would do well, particularly after looking up the convolutional nets that have done well on similar tasks. Without all the trimmings just this quickly takes the validation accuracy to over 93%. From there batch normalisation and dropout had the biggest impact enabling me to reach aver 94% accuracy. The choice of optimiser (Adam or Nadam), L2 regularisation, or which initialisations used (Keras default or the ones chosen) - didn't really make a huge difference most seemed sensible.

What helped bump up the result from here was data augmentation. Ensembling the models was not really necessary and the model achieved 95.4% accuracy without it. Being honest it was just something I wanted to see how to implement and it did no harm. The implementation list is given below.

- Ensembling: Three deep convolutional neural networks with fully connected layers
- Data Augmentation
- Nadam Optimiser
- Dropout
- Batch Normalisation
- Relu activations
- L2 regularisation
- He_normal or uniform initialisations
- Early stopping

I'm sure you trust me - but in case not! I have attached photo of my laptop screen showing the validation accuracy of 0.954 at the end of this document. Not that this was before ensembling which is not strictly necessary to get above 95%. See figure 5.

3 Exploding and Vanishing Gradients Qu 3

Task. Derive a bound for the growth of the norm of the gradient of the cost function with respect to the neural network weights. In the case of weight initialisation with unitary matrices.

We know that $W^{\dagger}W = WW^{\dagger} = I$, and $\|Wx\|_2 = \|x\|_2$ and require $\|\frac{\partial C}{\partial h_t}\|$

$$\|\frac{\partial C}{\partial h_t}\| = \|\frac{\partial C}{\partial h_n} D_n W_{n-1}^{\dagger} D_{n-1} W_{n-2}^{\dagger} \dots D_{t+1} W_t^{\dagger}\|$$

Where $D_{k+1} = \text{diag}(\sigma^{'}(W_k h_k + b_k)$

Therefore,

$$\|\frac{\partial C}{\partial h_t}\| \leq \|\frac{\partial C}{\partial h_n}\| \|D_n W_{n-1}^{\dagger} D_{n-1} W_{n-2}^{\dagger} \dots D_{t+1} W_t^{\dagger}\| = \|\frac{\partial C}{\partial h_n}\| \prod_{k=t}^{n-1} \|D_{k+1} W_k^{\dagger}\|$$

Which implies that

$$\|\frac{\partial C}{\partial h_t}\| \leq \|\frac{\partial C}{\partial h_n}\| \prod_{k=t}^{n-1} \|\text{diag}(\sigma^{'}(W_k h_k + b_k) W_k^{\dagger}\|$$

So using the unitary matrix property,

$$\|\frac{\partial C}{\partial h_t}\| \leq \|\frac{\partial C}{\partial h_n}\| \prod_{k=t}^{n-1} \|\text{diag}(\sigma^{'}(W_k h_k + b_k)\|$$

And if we set $\left|\sigma^{'}(W_k h_k + b_k)_j\right| = (\tau_k)_j$ (note the question notation seems not quite right at this point, so I used my own). Then we have that,

$$\|\frac{\partial C}{\partial h_t}\| \leq \|\frac{\partial C}{\partial h_n}\| \prod_{k=t}^{n-1} max_{j=1,\dots,n}(\tau_k)_j$$

And if we let $max_{j=1,\dots,n}\tau_k = \tau$, (we are taking the max over all the τ's). So finally we have that,

$$\|\frac{\partial C}{\partial h_t}\| \leq \|\frac{\partial C}{\partial h_n}\| \tau^{n-t}$$

From this we can see that gradients will tend closer to zero for $0 \leq \tau < 1$ and tend to get larger and potentially explode when $\tau > 1$. Relu is nice because the gradient is either 1 or 0 and thus aids the problem of vanishing or exploding gradients.

4 Audio hot keyword recognition Qu 4

Recall in part of Lecture 5, we examined a representative piece of code for training a hot keyword recognition model. An interesting aspect of the design was how each mini-batch was formed. Each batch was not just simply comprised of labeled training data containing examples of spoken words, but would include data that was altered by some transformation (e.g., zero-padding) or even data that did not even contain spoken words at all (e.g., unknown class).

Task. Select three techniques from the list at the end of this question. For each of these techniques provide a separate relatively brief paragraph that: (i) describes the technique and why it can help model robustness, (ii) discusses if and how it would be applicable outside the context of an audio model. Finally provide paragraph (include a diagram if this is helpful) that describes how mini-batch generation occurs within hot keyword recognition implementation we examined in class

4.1 Data Augmentation

4.1.1 Description

Data augmentation is a method of expanding our training set whilst maintaining the same semantic meaning. Indeed I used precisely this method to help me get over the 95% accuracy on the validation set for the NotMNIST task given I had such a small training set (13,000 examples).

The idea is simple - shift, shear, rotate our training 'images' in small ways whilst maintaining the same target label y. For example, an image of a cat moved slightly is still a cat, or flipped or sheared (within limits). By doing this we can enhance the robustness of our model by increasing the number of training examples. One has to be careful that one does this whilst maintaining the fidelity of our target 'y's.

4.1.2 Use outside Audio

I have already described a use outside audio. This is commonly used in vision tasks and used in this assignment on the NotMNIST task. For the audio task, and interesting aspect of the lecture model implementation is turning audio into an image through the use of a spectrogram. Thus we turn audio into an image and utilise a convolutional neural net to classify.

The data augmentation done in this case involves random time shifting, and stretching. Doing the same for pitch and also cropping and scaling images. Random jitter is also added but I will describe this in the subsection on background noise. An alternative reference to the lectures may be found in the paper 'Deep convolutional neural networks and data augmentation for environmental sound classification' by Salamon and Bello, [3].

4.2 Zero Padding

4.2.1 Description

If one imagines a convolution with a kernel over an image then zero padding simply adds a border of zeros around this image before the convolution. The size of the border depends upon the kernel one uses, but when using the 'same' option for zero padding in tensorflow, then the main output will be a convolved image with the same dimensions as the original.

- It can help preserve features at the edge of the image, thus improving performance.
- As mentioned above, it controls the size of the output feature map.

If one does not do this then the volume of the 'image' will reduce as one flows through the convolutional neural network and borders will be 'washed away'. In the model presented in lectures and the code provided, zero padding is utilised by choosing the 'same' option in tensorflow, so this is used in our audio application.

4.2.2 Use outside Audio

It is of course commonly used for images on convolutional neural networks, not just in audio. The tensorflow notes are found in the following link. `https://www.tensorflow.org/api_guides/python/nn#Notes_on_SAME_Convolution_Padding`.

4.3 Background Noise

4.3.1 Description

This one is reasonably self-evident. Noise is added is some form to the original training data. in the case of audio, this could be white noise, or indeed as described in the tensorflow notes on simple audio recognition `https://www.tensorflow.org/versions/master/tutorials/audio_recognition` a more sophisticated format such as commonly found background noises one might find in day to day life for audio.

The logic is to make the neural network predictions robust under varying noise quality and in different backgrounds. Again in the implementation described in class background jitter as well as background noises were added. (Selected in the code through the flag use_background_data).

4.3.2 Use outside Audio

Background noise is also used in applications outside data. Again this is common in vision applications. Indeed if one looks at a different kind of neural net - a denoising auto-encoder takes images, noises them up and outputs a clean image (the cost function is based on the original target image), this enables the denoising of noisy images. The effect of background noise on convolutional neural networks is still an area of research, for example the paper - 'Understanding How Image Quality Affects DeepNeural Networks' by Dodge and Karam [1].

4.3.3 Mini Batch generation in model used in lectures

The neural network for audio described in the lectures is a deep convolutional neural network using data augmentation as described above, background noise and zero padding for the convolutions and the max pooling, of course prior to anything audio is converted into 'images' as described by using a spectrogram.

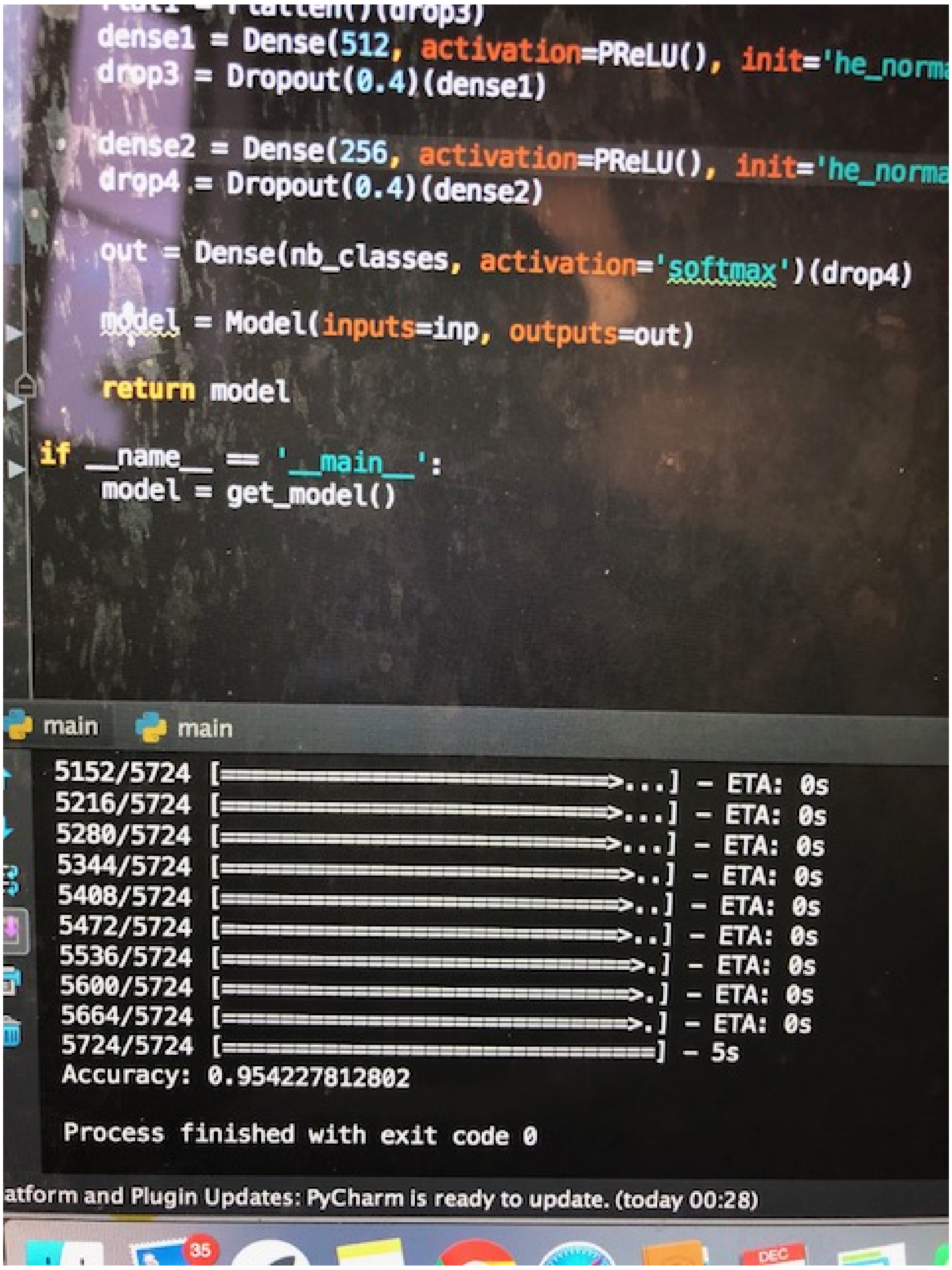
The various options for batch creation can be found in the segment of code given below. In this code time shifting is enabled, time shift padding, the use of background data and indeed a silence option where the background data is all that remains. Other additions described in the lectures were class balancing and amplitude scaling and control.

```
def get_batch_data(self,
                   sess,
                   batch_size=100,
                   feature_dimension=105 * 40,
                   offset=0,
                   time_shift=0,
                   audio_samples=16000,
                   background_frequency=0.8,
                   background_volume=0.1,
                   mode='training',
                   verbose=False):
        """
        Generates (batch) data for training/validation/testing

        Args:
            sess:                   TensorFlow session that was
                                            active when audio
                                            processor was created
            batch_size:             Target number of audio files
            feature_dimension:      Number of features extracted
                                            from audio frame
            offset:                 Where to start within the list
                                            while fetching audio
                                            data
            time_shift:             How much to randomly shift the
                                            clips by in time
            audio_samples:          Number of audio samples in an
                                            window
            background_frequency:   Probability of adding
                                            background noise to
                                            audio
            background_volume:      How loud the background noise
                                            will be
            mode:                   Which audio file partition to
                                            use. Must be in
                                    ['training',
                                    'testing',
                                    'validation']
            verbose:                Verbosity of the batch
                                            processing, True to
                                            print intermediate
                                            results

        Returns:
            batch_data:             Batch data preprocessed
                                            containing audio
                                            measurements
            batch_labels:           Labels for the batch data in
                                            one-hot form

        """
```



```
    dense1 = Dense(512, activation=PReLU(), init='he_norma
    drop3 = Dropout(0.4)(dense1)

    dense2 = Dense(256, activation=PReLU(), init='he_norma
    drop4 = Dropout(0.4)(dense2)

    out = Dense(nb_classes, activation='softmax')(drop4)

    model = Model(inputs=inp, outputs=out)

    return model

if __name__ == '__main__':
    model = get_model()
```

```
5152/5724 [==========================>...] - ETA: 0s
5216/5724 [==========================>...] - ETA: 0s
5280/5724 [==========================>...] - ETA: 0s
5344/5724 [===========================>..] - ETA: 0s
5408/5724 [===========================>..] - ETA: 0s
5472/5724 [===========================>..] - ETA: 0s
5536/5724 [============================>.] - ETA: 0s
5600/5724 [============================>.] - ETA: 0s
5664/5724 [============================>.] - ETA: 0s
5724/5724 [==============================] - 5s
Accuracy: 0.954227812802

Process finished with exit code 0
```

atform and Plugin Updates: PyCharm is ready to update. (today 00:28)

Figure 5: Validation Accuracy for question 2

References

[1] Samuel Dodge and Lina Karam. Understanding how image quality affects deep neural networks. In *Quality of Multimedia Experience (QoMEX), 2016 Eighth International Conference on*, pages 1–6. IEEE, 2016.

[2] Yarin Gal, Jiri Hron, and Alex Kendall. Concrete dropout. *arXiv preprint arXiv:1705.07832*, 2017.

[3] Justin Salamon and Juan Pablo Bello. Deep convolutional neural networks and data augmentation for environmental sound classification. *IEEE Signal Processing Letters*, 24(3):279–283, 2017.