Artificial Neural Networks

Prof. Dr. Sen Cheng

Dec 9, 2019

Problem Set 10: Deep Neural Networks

Tutors: Filippos Panagiotou (filippos.panagiotou@ini.rub.de), Thomas Walther (thomas.walther@rub.de)

1. **Fully connected (dense) networks.** Build and train a fully connected network with 4 layers. Note that *keras* does not have an explicit representation of the input layer. The input is passed directly to the first hidden layer. Make use of the Keras documentation (keras.io).

   (a) Import the 'fashion_mnist' dataset from the 'keras' library and load it. The data set contains 60 000 images with a dimension of 28x28 pixels. Visualize a few images to get a feel for the data. Vectorize the images for the training and test set.

   (b) Note that the images are in greyscale. Scale of the pixel values appropriately to the range [0.0;1.0]. Why is this necessary? (Hint: Think about the backward pass and what it does to the weights.)

   (c) Build a Sequential network. Add two `Dense` layers, with 10 units each and ReLU activation functions. Add another `Dense` layer with 10 units and a softmax activation function. The sizes of the hidden layers are flexible, but the size of the output layer must remain at 10 units. Why?

   (d) Calculate by hand how many trainable parameters are in your network (include the bias terms). Note it down for later.

   (e) Compile (`.compile()`) the network using Stochastic Gradient Descent (SGD) as an optimization method, the `sparse_categorical_crossentropy` loss function, and `accuracy` as metrics.

   (f) Train the network for 3 epochs. Use the test set for validation. Note down the accuracy metric on the training and test sets. Define and compile new networks and train them a few times to get a feel for the training process.

   (g) If you did not code it explicitly, find out what was the value that you used in the learning rate. Change the learning rate to 0.005 and train the network for a few times.

   (h) Restore the learning rate to the default. Increase the number of neurons in the hidden layers to 64. Calculate the number of trainable parameters and compare it to the number of parameters in the models above. Also compare the accuracies to those of the smaller models.

   (i) Reduce the training data to 20.000 items. Train a new network and look at its accuracy. Why does the test set accuracy diverge from the training set accuracy? Does it matter for us?

   (j) Using the trained network, find all elements of the test set that the model predicts to be ankle boots (label= 9). Visualize the retrieved items. Use a display loop to do that. Stop the loop whenever an item has been classified incorrectly. Visually inspect the incorrectly classified items. Give a reason for those items being mis-classified as ankle boots.

2. **Convolutions.**

(a) Given two 'image' matrices:

$$\mathbf{I}_A = \begin{bmatrix} 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \end{bmatrix} \qquad \mathbf{I}_B = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

and two filter (kernel) matrices:

$$\mathbf{K}_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} \qquad \mathbf{K}_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$$

manually compute the 2d convolutions $\mathbf{K}_i * \mathbf{I}_i$ for $i = x, y$ and $j = A, B$ on paper. Assume that the image matrices are padded with the appropriate number of additional rows and colums, which are filled by repeating their current border (you can compare with 'nearest' method in 'scipy.ndimage.convolve').

(b) Inspect the results of the convolutions: What kind of features do the filter kernels extract?

3. **Convolutional Neural Networks (CNN).** Build and train a 6-layer CNN using the *keras* library.

(a) Proceed as in problem 1, except for the following two points:

i. Change the dimensionality of your data appropriately so that they can serve as input for a CNN, i.e. $28 \times 28 \times 1$.

ii. Instead of the two hidden dense layers, use two consecutive pairs of 2-d convolution (`Conv2D`) and max-pooling (`MaxPooling2D`) layers. Use `Conv2D` with 16 and 8 filters of size of 3x3 respectively, and ReLU activation function, and `MaxPooling2D` layers with a 2x2 receptive field.

(b) Compile and train the network as in problems 1f. Compare the final accuracies. Does the discrepancy between training and test error remain? What are the reasons for this?

(c) Calculate the number of trainable parameters. Compare with the number of parameters for the fully connected network above.

4. **BONUS.** Modify the network from problem 3 according to the following instructions and observe how the training time, the training accuracy and the test accuracy change.

(a) Remove the two `MaxPooling2D` layers and add a stride of 2 to the `Conv2D` layers. Train this network and compare epoch speed. Why the speedup?

(b) Use the ADAM optimizer in place of the SGD. Why the increase in accuracy?

Solutions

1. **Fully connected (dense) networks.**

   Do all necessary imports.

```
import numpy as np
import keras
from keras.layers import Dense, Dropout, Flatten, Conv2D, MaxPooling2D
from matplotlib import pyplot as plotDisplay
```

   (a) Import fashion-mnist

```
### ex 1a
# load data
fashion_mnist_data=keras.datasets.fashion_mnist.load_data()

# unpack data
((train_x, train_y), (test_x,test_y)) = fashion_mnist_data

# vectorize
train_x.resize(train_x.shape[0], train_x.shape[1]**2)
test_x.resize(test_x.shape[0], test_x.shape[1]**2)
```

   (b) Normalize.

```
# normalize
train_x = train_x / 255.0
test_x = test_x / 255.0
```

   We normalize between 0 and 1 to aid the convergence process. Recall from gradient descend that if the error is too large, our models will not be able to find a minimum. Further, normalization helps with the vanishing and exploding gradients issues.

   (c) Build a fully connected feed-forward network:

```
fc = keras.models.Sequential()
fc.add(Dense(units=10, activation='relu',input_shape=(784,)))
fc.add(Dense(units=10,activation='relu'))
fc.add(Dense(units=10,activation='softmax'))
```

   The dimensionality of the final layer is 10 because we want to classify items in 10 classes (and because our labels are one-hot encoded).

   (d) Calculate the weights $(784 \times 10 + 10) + (10 \times 10 + 10) + (10 \times 10 + 10) = 8070$.

   (e) Compile the fully connected feed-forward network:

```
fc.compile(optimizer='SGD', loss='sparse_categorical_crossentropy',metrics=['accuracy'])
```

   (f) Train the network for 3 epochs:

```
batch_size = 64
epochs = 3
fc.fit(x=train_x, y=train_y, epochs=epochs, validation_data=(test_x, test_y),batch_size=batch_size)
```

   Training accuracy ≈ 0.768 and testing accuracy ≈ 0.771.

   (g) Learning rate: The default learning rate for the SGD optimizer in Keras is 0.01.

```
### ex 1g
fc2 = keras.models.Sequential()
fc2.add(Dense(units=10, activation='relu',input_shape=(784,)))
fc2.add(Dense(units=10,activation='relu'))
fc2.add(Dense(units=10,activation='softmax'))
sgd = keras.optimizers.SGD(lr=0.005) # change the learning rate
fc2.compile(optimizer=sgd, loss='sparse_categorical_crossentropy',metrics=['accuracy'])
fc2.fit(x=train_x, y=train_y, epochs=epochs, validation_data=(test_x, test_y),batch_size=batch_size)
```

Training accuracy ≈ 0.731 and testing accuracy ≈ 0.732.

(h) Increase number of hidden layer neurons The total number of parameters for this network is $(784 \times 64 + 64) + (64 \times 64 + 64) + (64 \times 10 + 10)) = 55050$! The following code produces an appropriate model:

```
### ex 1h
fc3 = keras.models.Sequential()
fc3.add(Dense(units=64, activation='relu',input_shape=(784,)))
fc3.add(Dense(units=64,activation='relu'))
fc3.add(Dense(units=10,activation='softmax'))
sgd = keras.optimizers.SGD(lr=0.01)
fc3.compile(optimizer=sgd, loss='sparse_categorical_crossentropy',metrics=['accuracy'])
fc3.fit(x=train_x, y=train_y, epochs=epochs, validation_data=(test_x, test_y),batch_size=batch_size)
```

Training accuracy ≈ 0.818 and testing accuracy ≈ 0.818.

(i) Reduce training data

```
# reduce size of dataset
train_x=train_x[:20000]
train_y=train_y[:20000]
```

Then we compile a new model and train it on the data. The accuracy of the model on the test set should start falling below the training accuracy (e.g. training accuracy ≈ 0.767 vs test accuracy ≈ 0.752). This phenomenon is called overfitting, and it happens because the network is large enough that it can memorize the training examples. This is problematic, because we are interested in predicting items that are not in the training dataset.

(j) Visualize all items in the test set that the model predicts to be ankle boots in a display loop:

```
predictions = np.argmax(fc.predict(test_x), axis=1)

for i in range(predictions.shape[0]):
    if predictions[i] == 9:
        plotDisplay.imshow(test_x[i, :, :, 0])
        plotDisplay.pause(0.01)

        if test_y[i]!=9:
            print('WRONG classification! True class of this item is %d.' % test_y[i])
            plotDisplay.show()
        else:
            print('Item correctly classified!')
        plotDisplay.cla()
```

Items that are misclassified look somewhat similar to ankle boots.

2. **Convolutions.**

(a)

$$\mathbf{K}_x * \mathbf{I}_A = \begin{bmatrix} 0 & -4 & 0 & 4 & 0 \\ 0 & -4 & 0 & 4 & 0 \\ 0 & -4 & 0 & 4 & 0 \\ 0 & -4 & 0 & 4 & 0 \\ 0 & -4 & 0 & 4 & 0 \end{bmatrix} \qquad \mathbf{K}_y * \mathbf{I}_A = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

$$\mathbf{K}_x * \mathbf{I}_B = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix} \qquad \mathbf{K}_y * \mathbf{I}_B = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ -4 & -4 & -4 & -4 & -4 \\ 0 & 0 & 0 & 0 & 0 \\ 4 & 4 & 4 & 4 & 4 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

(b) The filter kernels can be used to detect vertical (K_x) and horizontal (K_y) edges in images (see 'Sobel-Filter' for more information).

3. **Convolutional Neural Networks (CNN).**

```
# reshape for CNN
train_x = train_x.reshape(train_x.shape[0],28,28, 1)
test_x = test_x.reshape(test_x.shape[0],28,28, 1)
input_shape = (train_x[0].shape[0],train_x[0].shape[1], 1)

CNN = keras.models.Sequential([
Conv2D(filters=16,kernel_size=(3,3), activation='relu', input_shape=input_shape),
MaxPooling2D(pool_size=(2,2)),
Conv2D(filters=8,kernel_size=(3,3), activation='relu', input_shape=input_shape),
MaxPooling2D(pool_size=(2,2)),
Flatten(),
Dense(units=10,activation='softmax')
])
```

(a) Compile the network:

```
CNN.compile(optimizer='SGD', loss='sparse_categorical_crossentropy', metrics=['accuracy'])
```

Train the network.

```
CNN.fit(train_x, train_y, epochs=3)
```

Indicatively our architecture achieved an accuracy of 0.8553 and a validation accuracy of 0.8577 with three epochs of training.

(b) Convolutional neural networks have an implicit architectural bias for local connections that comes from performing the convolution operation. This implicit bias seems to let them perform and generalize better on image processing tasks, where spatially close information is important.

(c) Calculate number of weights

Convolutional layers are light on parameters, and max pooling layers do not have any. The first layer has 16 filters of size 3×3 plus their biases for a total of $3x3x16 + 16 = 160$ parameters. The second layer receives an input from each of the 16 filters in the first layer and applies eight 3×3 filters, so it has $3 \times 3 \times 16 \times 8 + 8 = 1160$ parameters, which brings the total so far to 1320.

Most of the trainable parameters for this model actually come from the dense layer at the end. Calculating the dimensionality of the dense layer is tricky because we have to track the dimensionality of the input

as it changes when it is propagated through the network. The first convolutional operation returns a $26 \times 26 \times 16$ output (downgraded from 28×28 because we did not use padding). The max pooling operation halves that to $13 \times 13 \times 16$ which is given as input to the next convolutional layer, which also doesn't use padding. So the dimensionality of the output becomes $11 \times 11 \times 8$. Max pooling reduces it to $5 \times 5 \times 8$ that we map on the 10 fashion-mnist classes. So the weights for the neural network are $200 \times 10 + 10 = 2010$. This brings the total number of trainable weights in this convolutional neural network to $1320 + 2010 = 3330$ in total.

4. **BONUS.**

(a) Strided convolutions

```
CNN2 = keras.models.Sequential([
Conv2D(filters=16,kernel_size=(3,3), activation='relu', input_shape=input_shape,strides=(2,2)),
Conv2D(filters=8,kernel_size=(3,3), activation='relu', input_shape=input_shape,strides=(2,2)),
Flatten(),
Dense(units=10,activation='softmax')
])

CNN2.compile(optimizer='SGD', loss='sparse_categorical_crossentropy',metrics=['accuracy'])
CNN2.fit(x=train_x, y=train_y, epochs=epochs, validation_data=(test_x, test_y), batch_size=batch_size)
```

The total number of trainable weights on this network is $3 \times 3 \times 16 + 16) + (3 \times 3 \times 16 \times 8 + 8) + (288 \times 10 + 10) = 4210$. The difference is because we round down odd-number dimensions one time less than in the previous model. Pooling is a cheaper operation than a convolution, as it only needs to make a comparison between the numbers in the current filter area. However, if we use a stride during the convolution operation then we do not have to do as many convolution operations, which results in faster performance of strided convolution despite the increase in the number of trainable parameters.

(b) ADAM optimizer

```
CNN3 = keras.models.Sequential([
Conv2D(filters=16,kernel_size=(3,3), activation='relu', input_shape=input_shape,strides=(2,2)),
Conv2D(filters=8,kernel_size=(3,3), activation='relu', input_shape=input_shape,strides=(2,2)),
Flatten(),
Dense(units=10,activation='softmax')
])

CNN3.compile(optimizer='adam', loss='sparse_categorical_crossentropy',metrics=['accuracy'])
CNN3.fit(x=train_x, y=train_y, epochs=epochs, validation_data=(test_x, test_y), batch_size=batch_size)
```

The ADAM optimizer uses an adaptive learning rate that starts off large as the error is large, and decreases with the error. At the same time, it stabilizes the gradient trajectory by discounting large perturbations in the gradients, predominantly in the bias updates. This makes the learning process smoother and faster. The method is a weighted combination of two other gradient descend algorithms, Momentum SGD and RMSProp.