AMATH 482 Winter 2020

Homework 5: MNIST Fashion Classification

Tanner Graves

Introduction and Overview

Abstract

The following documents a first exploration of neural networks for image classification. Using the MNIST Fashion dataset, which consists of 50000 28 by 28 images of 10 different categories of clothing items, we will aim to build neural networks both dense and convolutional to classify these images into their respective categories. Additionally, we will explore popular architectures and methods in search of more effective networks.

Theoretical Background

Neural networks are a type of machine learning in which large sets of nodes not too dissimilar to neurons in a brain are trained in their connection to one another to complete a task. The most basic of neural networks is a dense, or fully connected network. This refers to every node in a layer of the network being connected to every single node of the consecutive layers. Each of these nodes has a value and each connection a weight. For each node in the next layer the sum of all other nodes including a constant bias term is multiplied by their respective weight and then fed through an activation function, typically a hyperbolic tangent function or a rectified linear unit (ReLU) which is just the max of the input and zero. For the application of classification into discrete categories, the final layer will consist of a node for each category and maximization function that incentivizes decisive output.

The key to the network's functionality is training, or fitting. During this process the values of the weights are optimized through a process called back propagation. The process can be understood in terms of thinking of the collection of weights as a vector. We then need a loss function, or something we want to minimize. After moving in a direction we may calculate the change in this objective function with respect to each component of the vector. This is the gradient, or the direction of greatest increase of the function. Since we aim to minimize this function we will head opposite this direction, calculate a new gradient, and continue iterating. Modern processes will ultimately work similarly albeit a bit more cleverly.

An alternative layer structure exists called a convolutional layer, that can be thought to work on images as opposed to generalized nodes. The output of each convolutional layer will be a 3D tensor of size determined by the height, width, and depth(number of filters or kernels) of the convolution. This can be thought of as a set of feature maps, or processed images. A kernel is a matrix and can be thought to be a filter that is applied to a subset of an input image and translated across each dimension. As we translate across our input image, we take the inner product of the kernel and the subset of the input image. The result of one iteration of this is one entry in an output feature map. Each convolutional layer will have several different kernels which will output several different feature maps.

It is important to observe that as the filter is translated across either all the rows or all the columns of the input space, either the thickness or the height of the filter would prevent filters from being centered on border entries. This results in a reduction of size of the output space. Under these assumptions, given a convolutional layer of a neural network with input space dimension (m_1, n_1, d_1) and desired kernel dimension (m_2, n_2) and number of filters d_2, the output space will be of dimension:

$$(m_1 - m_2 + 1, n_1 - n_2 + 1, d_2)$$ Figure 1

We may prevent this shrinking phenomenon by the practice of 'padding' where the original matrix is surrounded in zeros, allowing the convolution process to output a matrix of size (m_1, n_1).
The output dimension of a convolutional layer is also affected by the 'stride' of a filter. Whereas the normal filtering process would have us translating a kernel by 1 in a particular direction, a convolution layer with a stride of n would translate n units in a particular direction per translation. This results in fewer total possible translations along each dimension of the input space and therefore, a smaller output space.

It is possible that a powerful enough network will start overfitting the training data. That is the network will be trained to be sensitive to patterns that exist only in the training data, and is not useful for the generalized task of classification. Dropout is a means of preventing overfitting in larger neural networks. It is implemented as a layer and supplied with a probability that a given node will be excluded from changing during any iteration of the training process. This does, however, result in increased training time. Overfitting may be prevented alternatively via regularization: a process where having particularly high weights are decentiviced during training.

Research Results

Many successful convolutional neural networks employ convolutional layers at the bottom of a network in a series of stages with a couple of dense layers at the top. Convolution has been found to be a process very well suited for image processing, at each layer extracting increasingly abstract feature maps where dense layers have been well suited for classification. For any given convolutional stage only feature maps of a particular size will be considered. These feature maps will be fed through often two or three convolutional layers before being pooled to a smaller size and being fed forward to the next stage of the network. This results in a sort of pyramid structure, where the image is processed into progressively smaller feature maps throughout the network. As the feature maps become smaller, it becomes less computationally prohibitive for training to increase the number of kernels. That said, successful networks will have the number of connections per layer increasing roughly exponentially through the network.

Despite me being initially skeptical about the usefulness of padding, as I didn't think that the edges of images contributed much useful information in this dataset, it turns out that its practicality is two-fold: it prevents an unnecessary reduction of the output space at each layer with no extra information given, and it improves accuracy.

Data augmentation is a practice where training images are distorted (via translation, rotation, skew, flipping, etc.) to force the network to pick up on more relevant features of a particular class that are invariant under augmentation). This method has been shown to improve a CNNs accuracy on the fashion mnist data set by as much as 1%.

Reading resulted in some miscellaneous discoveries that influenced my network development like: it is found that ReLU activation results in quicker training and Adam is an improvement over SGD(Stochastic Gradient Descent) and other similar trainers. Additionally I found an alternative philosophy to training. Using K-folds to make a series of splits between test and validation data, and fitting using each allows for a better understanding of what a networks average accuracy is.

Algorithm Implementation and Development

When attempting to mimic some notorious CNNs, I was immediately struck by the sheer size of these networks. Image net has 60 million parameters and VGG 138 million! Since I am armed only with an old think pad, and I didn't much enjoy staring at networks train for very long, I put a great deal of effort into how well I could get a network to perform while also trying to minimize the number of trainable parameters.

Realizing that overall accuracy was not a sensitive means of testing the performance of a network, I quickly turned to looking at the confusion matrix of the training data. This provided many more metrics with which to measure the changes to my network.

Much of the literature I could find around neural nets for image classification credited ReLU training faster than any of the sigmoid functions, so I quickly made the decision to switch to it.

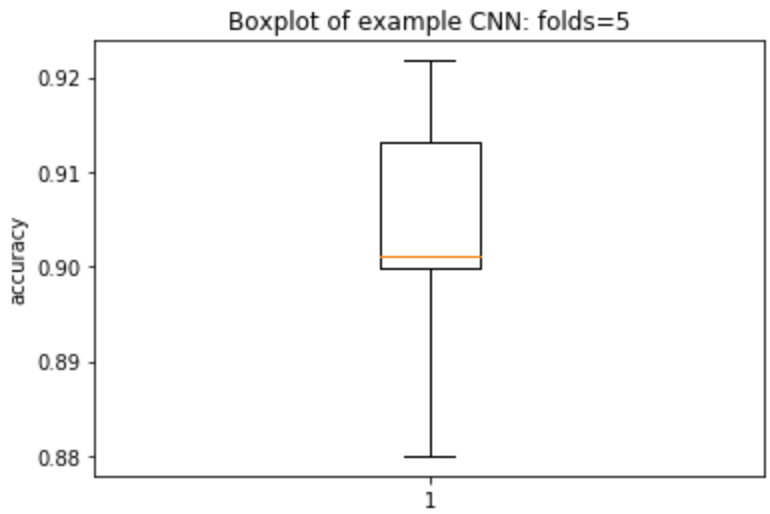
The final CNN that I ended up using was heavily influenced by VGG. That is, I would have stages of a couple of convolutional layers then being pooled down and fed forward into a next stage.

At no point was it apparent that any of my models were overfitting. In fact more often than not the validation accuracy was higher than the training accuracy. As a result I did foresee any benefit of introducing dropout or increasing regularization on any of my models. I anticipate that this is something that would be more of a serious issue as I deal with larger networks that are more capable of fitting to minute patterns specific to the test data.

I attempted to implement data augmentation, however, didn't follow the idea long due to the tools available in keras for this being somewhat inflexible. Augmentation may be easily achieved with the ImageDataGenerator object. However, this generator then formats training data in a specific way that increases training time significantly. I was attempting to use the example CNN as a benchmark to measure the difference data augmentation could make, but the fitting took around 30 minutes.

Computational Results

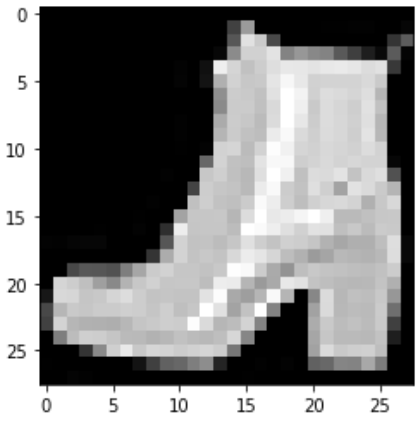
Upon changing hyperparameters for any amount of time it becomes immediately clear that the validation accuracy of the network is not overly sensitive. Wildly changing hyperparameters(eg. Increasing the width of a network ten-fold, or the depth three-fold) would in most cases yield miniscule changes in validation accuracy. Additionally, the accuracy of a neural net is subject to stochastic processes that introduces variance in accuracy with constant parameters. These variances were found to be fairly large, and are likely to have overshadowed changes caused by varying hyperparameters. The following figure shows that the example CNN code with a 5-fold test varied as much as 4% over all tests!



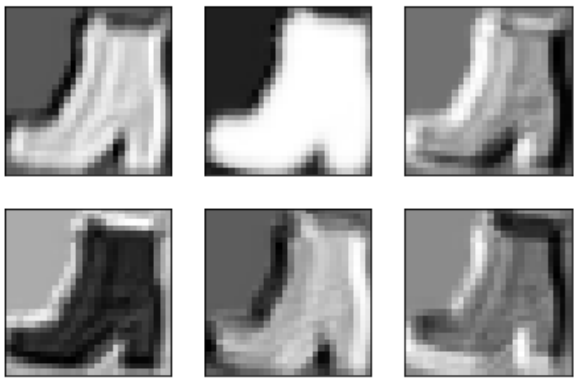
Using K-folds to get a better idea about the performance of a network was nice. However, it requires that fitting be done K-times to get the additional data points, so it was forgone on most runs. However, it served as a valuable illustration of the variability that exists in the output of networks when you do so little training (consistently 5 epochs).

An important step in understanding convolutional networks was to understand the output of each lay as a set of feature maps. It was possible to dissect my network to get a visualization of what is being output from a convolutional layer. Below is an example of what is output by the first convolutional convolutional layer of a network with a 3 by 3 kernel size. When given a picture of a boot (left) the convolutional layer was able to extract the following feature maps (right)

Input image



Feature maps out from layer

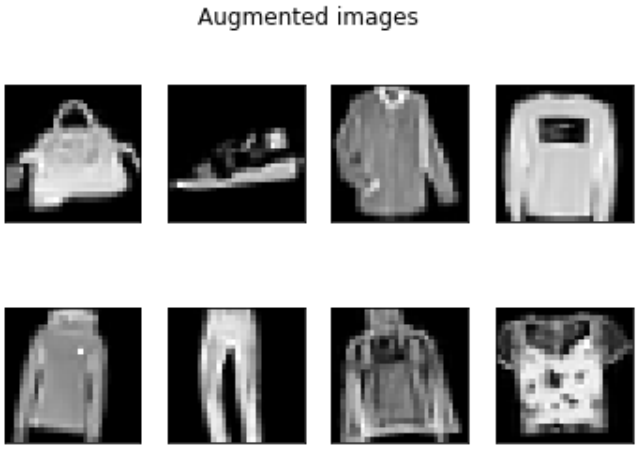


In this example we can begin to see some of the features fitting was optimizing for. In some the foreground is highlighted, where others the background. Across the features maps we can see that a variety of edges have been highlighted.

Training the example CNN with augmented data gave test accuracy of 88.75%. The data augmentation parameters were as follows with some examples of augmented images:

Maximum rotation:	8°

Maximum shift:	8%
Maximum shear:	30%
Maximum zoom:	8%



My attempts at making a CNN in the spirit of VGG and a fully connected network resulted in the following architecture:

Convolutional Network

Layer	Type	Output Shape	Number of Parameters
1	Convolutional	(28,28,8) w=3	80
2	Convolutional	(28,28,16) w=5	3216
3	Max Pooling	(14,14,16)	
4	Convolutional	(14,14,16) w=3	2320
5	Convolutional	(14,14,32) w=5	100384
6	Max Pooling	(7,7,32)	
7	Convolutional	(7,7,32)	9248
8	Convolutional	(1,1,32) w=7 (no padding)	50208
9	Flatten	(32)	
10	Dense	(84)	2772
11	Out (softmax)	(10)	850

Dense Network

Layer	Nodes	Number of Parameters
1	784	
2	500	392500
3	500	250500
4	500	250500
5	10 (out)	5010

For the fully connected networks, I found that making the networks too deep(usually more than 3 layers) would often result in overfitting. Varying hyperparameters would seldom get the validation

accuracy over 88% to 89% consistently. The given dense network was able to achieve a test accuracy of 89.52%, and the given convolutional network got a test accuracy of 90.86%. The networks performance is summarized by the following confusing matrices that charts the category guessed against the correct category:

CNN Confusion Matrix

	0	1	2	3	4	5	6	7	8	9
0	807	1	14	18	2	2	148	0	8	0
1	2	983	0	11	2	0	2	0	0	0
2	12	1	910	8	38	0	30	0	1	0
3	10	7	10	915	29	0	26	0	3	0
4	1	0	77	27	855	0	37	0	3	0
5	0	0	0	0	0	993	0	5	0	2
6	86	0	79	23	64	0	737	0	11	0
7	0	0	0	0	0	16	0	978	0	6
8	4	1	1	4	2	2	0	2	984	0
9	1	0	0	0	0	8	0	69	0	922

Dense Confusion Matrix

	0	1	2	3	4	5	6	7	8	9
0	885	2	7	28	2	0	70	0	6	0
1	2	977	1	14	1	0	5	0	0	0
2	24	0	830	10	83	0	53	0	0	0
3	25	1	11	932	15	0	13	0	3	0
4	0	1	98	45	807	0	47	0	2	0
5	0	0	0	1	0	967	0	13	0	19
6	148	0	86	25	64	0	671	0	6	0
7	0	0	0	0	0	25	0	952	0	23
8	6	0	2	6	1	3	3	7	972	0
9	0	0	0	0	0	4	1	36	0	959

Summary and conclusion

There are many takeaways from this first exploration of neural networks. Firstly, it is nice to see how easy it was to setup the architecture of a neural network. With an understanding of basic theory and programming keras makes machine learning with neural networks incredibly accessible.

The findings of this project show that a neural network as a solution to classification problems can be wildly diverse in architecture and deliver similar results. It was shown, however, that convolutional networks are significantly better equipped for the task of image processing, with them capable of appreciably higher accuracy than networks that consist of dense layers alone. In researching networks, I additionally came across numerous tools to improve a networks accuracy, training speed, efficiency, and overfitting mitigation.

References

https://machinelearningmastery.com/how-to-develop-a-cnn-from-scratch-for-fashion-mnist-clothing-classification/

https://machinelearningmastery.com/how-to-visualize-filters-and-feature-maps-in-convolutional-neural-networks/

https://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf

https://static.googleusercontent.com/media/research.google.com/en//pubs/archive/43022.pdf

https://github.com/khanhnamle1994/fashion-mnist/blob/master/CNN-3Conv.ipynb

Appendix A. MATLAB Functions

```
models.Sequential()
partial()-convenient tool allows us to create aliases of functions
with commonly used params
layers.Flatten()-turns multi-dim tensor into a vector(for dense
layers)
layers.Dense()
layers.Conv_2D()
layers.MaxPooling2D()-used to decrease the image dim of output space
layers.BatchNormalization()-normalizes the input of a layer
layers.Dropout()-creates a probability that any given param will be
excluded from an iteration of training
```

```
model.compile()
model.fit()-used to initialize the training of a network
ImageDataGenerator()-used for input augmentation
```

Appendix B. python Code and Data

```
import numpy as np
import tensorflow as tf
import matplotlib.pyplot as plt
import pandas as pd
from sklearn.metrics import confusion_matrix
# In[3]:
fashion_mnist = tf.keras.datasets.fashion_mnist
(X_train_full,y_train_full), (X_test,y_test) =
fashion_mnist.load_data()
# In[4]:
X_valid = X_train_full[:5000] / 255.0
X_train = X_train_full[5000:] / 255.0
X_test = X_test / 255.0
y_valid = y_train_full[:5000]
y_train = y_train_full[5000:]
X_train = X_train[..., np.newaxis]
X_valid = X_valid[..., np.newaxis]
X_test = X_test[..., np.newaxis]
# In[5]:
from functools import partial
my_dense_layer = partial(tf.keras.layers.Dense,
activation="tanh",kernel_regularizer=tf.keras.regularizers.l2(0.0001)
)
my_conv_layer = partial(tf.keras.layers.Conv2D,
activation="tanh",padding="valid")
example = tf.keras.models.Sequential([
    my_conv_layer(6,5,padding="same",input_shape=[28,28,1]),
    tf.keras.layers.AveragePooling2D(2),
    my_conv_layer(16,5),
    tf.keras.layers.AveragePooling2D(2),
    my_conv_layer(120,5),
    tf.keras.layers.Flatten(),
    my_dense_layer(84),
    my_dense_layer(10, activation="softmax")
])
# In[38]:
from functools import partial
```

```
my_dense_layer = partial(tf.keras.layers.Dense,
activation="relu",kernel_regularizer=tf.keras.regularizers.l2(0.0001)
)
my_conv_layer = partial(tf.keras.layers.Conv2D,
activation="relu",padding="same")
myModel = tf.keras.models.Sequential([
    my_conv_layer(8,3,input_shape=[28,28,1]),
    my_conv_layer(16,5),
    tf.keras.layers.MaxPooling2D(2),
    my_conv_layer(16,3),
    my_conv_layer(32,14),
    tf.keras.layers.MaxPooling2D(2),
    my_conv_layer(32,3),
    my_conv_layer(32,7,padding="valid"),
    tf.keras.layers.Flatten(),
    my_dense_layer(84),
    my_dense_layer(10, activation="softmax")
])
myModel.summary()
# In[39]:
myModel.compile(loss="sparse_categorical_crossentropy",
             optimizer=tf.keras.optimizers.Adam(learning_rate=0.001),
             #optimizer = tf.keras.optimizers.SGD(lr=0.01,
momentum=0.9),
             metrics=["accuracy"])
# In[57]:
history = big.fit(X_train,y_train,
epochs=5,validation_data=(X_valid,y_valid))
# In[24]:
from sklearn.model_selection import KFold
##K-fold trainer
mdl = example
dataX = X_train_full
dataY = y_train_full
dataX = dataX[..., np.newaxis]
n_folds = 5
scores, histories = list(),list()
kfold = KFold(n_folds, shuffle=True, random_state=1)
for train_ix,valid_ix in kfold.split(dataX):
    mdl.compile(loss="sparse_categorical_crossentropy",

#optimizer=tf.keras.optimizers.Adam(learning_rate=0.001),
```

```
            optimizer = tf.keras.optimizers.SGD(lr=0.01,
momentum=0.9),
            metrics=["accuracy"])
    trainX, trainY, validX, validY = dataX[train_ix],
dataY[train_ix], dataX[valid_ix], dataY[valid_ix]
    history = mdl.fit(trainX,trainY,epochs=10,
batch_size=32,validation_data=(validX,validY))
    _,acc = mdl.evaluate(validX,validY)
    scores.append(acc)
    histories.append(history)
# In[42]:
pd.DataFrame(history.history).plot(figsize=(8,5))
plt.grid(True)
plt.gca().set_ylim(0,1)
plt.show()
# In[52]:
y_pred = myModel.predict_classes(X_train)
conf_train = confusion_matrix(y_train,y_pred)
print(conf_train)
# In[44]:
myModel.evaluate(X_test,y_test)
# In[46]:
y_pred1 = myModel.predict_classes(X_test)
conf_test = confusion_matrix(y_test,y_pred1)
print(conf_test)
# In[51]:
fig,ax=plt.subplots()
fig.patch.set_visible(False)
ax.axis("off")
ax.axis("tight")
df = pd.DataFrame(conf_test)
ax.table(cellText=df.values,
rowLabels=np.arange(10),colLabels=np.arange(10),loc="center",cellLoc=
"center")
fig.tight_layout()
#plt.savefig("conf.mat.pdf")
# In[249]:
myModel.layers[0].name
filters, biases = model.layers[0].get_weights()
# normalize filter values to 0-1 so we can visualize them
f_min, f_max = filters.min(), filters.max()
filters = (filters - f_min) / (f_max - f_min)
# plot first few filters
```

```
n_filters = filters.shape[3]
ix = 1;
for i in range(n_filters):
    # get the filter
    f = filters[:, :, :, i]
    # plot each channel separately
    for j in range(1):
        # specify subplot and turn of axis
        ax = plt.subplot(n_filters, 30, ix)
        ax.set_xticks([])
        ax.set_yticks([])
        # plot filter channel in grayscale
        plt.imshow(filters[:, :,j,i], cmap='gray')
        ix += 1
# show the figure
plt.show()
# In[53]:
mdl = example
garbo = tf.keras.Model(inputs=mdl.inputs,
outputs=mdl.layers[0].output)
garbo.summary()
# load the image with the required shape
img = X_test[1510,:,:,:]
# expand dimensions so that it represents a single 'sample'
#img = expand_dims(img, axis=0)
img = img[np.newaxis,...]
# prepare the image (e.g. scale pixel values for the vgg)
#img = preprocess_input(img)
# get feature map for first hidden layer
feature_maps = garbo.predict(img)
# plot all 64 maps in an 8x8 squares
nrow = 2
ncol = 3
ix = 1
for _ in range(nrow):
     for _ in range(ncol):
          # specify subplot and turn of axis
          ax = plt.subplot(nrow, ncol, ix)
          ax.set_xticks([])
          ax.set_yticks([])
          # plot filter channel in grayscale
          plt.imshow(feature_maps[0, :, :, ix-1], cmap='gray')
          ix += 1
```

```
# show the figure
plt.show()
plt.imshow(img[0, :, :, 0],cmap="gray")
# In[31]:
from numpy import mean
from numpy import std
print('Accuracy: mean=%.3f std=%.3f, n=%d' % (mean(scores)*100,
std(scores)*100, len(scores)))
# box and whisker plots of results
plt.boxplot(scores)
plt.title("Boxplot of example CNN: folds=5")
plt.ylabel("accuracy")
plt.show()
# In[33]:
exampleScores=scores
# In[7]:
from tensorflow.keras.preprocessing.image import ImageDataGenerator
gen = ImageDataGenerator(rotation_range=8, width_shift_range=0.08,
shear_range=0.3,
                                 height_shift_range=0.08,
zoom_range=0.08)
batches = gen.flow(X_train, y_train, batch_size=256)
val_batches = gen.flow(X_valid, y_valid, batch_size=256)
gen.fit(X_train)
itr = gen.flow(X_train)
# In[9]:
history3 = example.fit(batches,steps_per_epoch=55000//256, epochs=50,
                    validation_data=val_batches,
validation_steps=5000//256)
# In[37]:
smplBatch = batches.__getitem__(210)[0]
smplBatch.shape
ix = 1
strt = 48
for imNum in range(strt,strt+8):
    ax=plt.subplot(2, 4, ix)
    ax.set_xticks([])
    ax.set_yticks([])
    plt.imshow(smplBatch[imNum,:,:,0],cmap="gray")
    ix = ix+1
plt.suptitle("Augmented images")
# In[39]:
model = tf.keras.models.Sequential([
```

```
    tf.keras.layers.Flatten(input_shape=[28, 28]),
    my_dense_layer(500),
    my_dense_layer(500),
    my_dense_layer(500),
    my_dense_layer(10, activation="softmax")
])
```