MATH 498: Foundations of Machine Learning, Winter 2022
Project 2: Multiclass Classification

Cooper Stevens (coopstev@umich.edu)
Yuxiang Ying (yingyx@umich.edu)
Sizhuang He (sizhuang@umich.edu)

April 22, 2022

Experimental Protocol

We aim to recognize (from 10 possible classes) the type of clothing in a 28×28 black and white image from the `fashion_mnist` data set. To do this, we utilize neural networks. We investigate the performance of multiple network architectures and the effects of changing their many hyperparameters. After training on the provided training set, the best-performing network architecture that we discovered yields an accuracy of approximately 0.9264 on the provided testing set.

1 Tools

Python = 3.7
numpy = 1.21.6
pandas = 1.3.5
tensorflow = 2.8.0

2 Algorithm

- The architecture of our model is: Input → Convolution Layer → BatchNormalization Layer → MaxPooling Layer → Convolution Layer → BatchNormalization Layer → MaxPooling Layer → a dense network → Output.
- We applied early stopping, drop out, and regularization to discourage overfitting.
- The loss function we use is categorical cross-entropy, defined by

$$CE(x) = -\sum_{i=1}^{10} y_i \log f_i(x)$$

- We use the Adam optimizer[1] which is a substitution for stochastic gradient descent that we've learned in class.
- The activation function we use is relu function

$$f(x) = \max\{0, x\}$$

  to non-linearize the model.
- Output activation function is Softmax, since it's multiclass classifier.

$$\sigma(z)_i = \frac{e^{z_i}}{\sum_{j=1}^{10} e^{z_j}}$$

3 Features

When it comes to feature transformations, we perform only one deterministic feature transformation for the final model, but we did try another in our exploration: depending on the architecture of the first layer in the network, we sometimes would have to transform the input to be of the correct shape. For networks without a convolution or MaxPooling layer, we would transform each 28×28 image into a 784-length array.

The feature transformation that we continue to use in our final model is that of normalizing the input so that all 784 values in the image are in the range $[0, 1]$. We did this by dividing all values by 255 which is the largest possible value of any pixel. Scaling all pixels to be in the range $[0, 1]$ is an intuitive transformation for this data as it is a black and white image, so a number in the range $[0, 1]$ can indicate the intensity of the color white in that pixel. Hence, no information is lost in doing this feature transformation. As discussed in class, this feature transformation is recommended so that, if we were to add additional dimensions to the architecture, they would all be of the same scale. Furthermore, normalizing generally speeds up learning and leads to faster convergence.[2]

We did, however, train many non-deterministic feature transformations in the form of convolution layers and MaxPooling layers. A convolution layer will take in an image of width W and height H with C channels where $W, H, C \in \mathbb{Z}^+$. A convolution layer has hyperparameters $F \in \mathbb{Z}^+$ the number of filters, kernel size $(X, Y) \in \mathbb{Z}^+ \times \mathbb{Z}^+$ which dictate the dimension of each filter as $X \times Y \times C$, and the stride $S \in \mathbb{Z}^+ \times \mathbb{Z}^+$. The layer also has hyperparamters relating to padding (how/if the layer will add dummy values to the output to maintain the input size of $X \times Y$), activation function for each computed value in the new channels, and a regularizer. We use SAME padding (which puts 0s on the edges of the output image to maintain the size to be $X \times Y$) and ReLU activation (which puts $\max\{0, x\}$ in place of the value computed x by a filter), and L_2 regularization with regularization constant 10^{-4}.

Each filter $f \in \mathbb{R}^{X \times Y \times C}$ defines a feature transformation in that (after flattening the filter and the computes a dot product ($+$ a bias term $b_f \in \mathbb{R}$) with the corresponding units in the input image to

[1] Kingma, D. P., Ba, J. (2017, Jan 30). *Adam: A Method for Stochastic Optimization*. arXiv.

[2] Stöttner, T. (2019, May 16). *Why Data should be Normalized before Training a Neural Network*. Medium. Retrieved April 22, 2022.

give a single entry in a new channel. Every time it computes an entry, it moves ahead by the stride S and computes an entry the same way for those corresponding units in the image. In doing so over the whole input image, a new channel is derived from the input, and this channel is stacked alongside the channels derived by other filters in the convolutional layer, essentially creating a new image that can be passed to the next layer in the network.

These F filters each have $X \times Y \times C + 1$ independent parameters that are learned as the model is trained. Hence, the feature transformations associated with a convolution layer are not deterministic, as they are determined by the randomized process of training (random in the ordering of the input images and in the dropout of connections in a later dropout layer).

After each convolutional layer, we have a MaxPooling layer. This is a feature transformation with hyperparameter window size $W \in \mathbb{Z}^+ \times \mathbb{Z}^+$, stride $S \in \mathbb{Z}^+ \times \mathbb{Z}^+$, and padding settings. We use VALID padding which adds no dummy values to the output, so output size is determined by the window size and the stride. A MaxPooling layer looks at a contiguous window of size W in a channel and takes the maximum of all values in the window. This value is placed into a new channel. Every time it computes an entry in the new channel, it moves the window ahead in the current channel by the stride S and computes an entry the same way, again placing the computed value into the new channel. Upon moving the window through the whole channel as dictated by the stride, it repeats for the next channel, stacking the new channels in turn to create yet another image (with the same number of channels as the input) as the output of the MaxPooling layer.

4 Parameters

Due to the black box nature of neural networks, we came up with the architecture and some hyperparameters of our final model with trial and error. It's hard to explain why our final model has a satisfying performance with test accuracy of approximately 0.92, so in this section we will compile a history of updates we made on our models and how these updates affect the performance.

Some tunable hyperparameters include: the architecture, number of layers in the dense network, number of neurons in each layer, choice of activation function, early stopping, patience, choice of optimizers, number of filters, and filter sizes in the convolution layer.

Our first model is the one from the lab section. It is a dense network with two hidden dense layers, each containing 64 neurons and the activation function is the ReLU function. This model reached a test accuracy of 0.8832.

In the second version, we changed activation functions to sigmoid and $\tanh$. This slightly reduced the test accuracy. The outcome is 0.8795.

Then, we restored the activation functions back to ReLU's and introduced early stopping with patience of 3. The outcome is 0.8702.

We found that the performance of dense networks were always around 0.88 accuracy, so we wanted to try something new. We added a MaxPooling layer. However, adding a MaxPooling layer alone sig-

nificantly reduced the performance of our model: the outcome is 0.8127. Hence, we added convolution layers before each of the MaxPooling layers: the outcome is 0.9009, which is the best so far.

Finally, we added batch normalization layers in between each convolution and pooling layers and added L_2 regularizers in each layer. This is our final model. The outcome is 0.9264.

When tuning the hyperparameters, we also tried early stopping with patience of 5 and the difference is insignificant. We tried using a Stochastic Gradient Descent optimizer, and this also reduces the performance.

In summary, significant improvements were mainly brought by the introduction of convolution layers and MaxPooling layers together. Dense networks that we tried could only achieve test accuracy of 0.88 at most, and the convolution layers together with MaxPooling boosted our model to 0.92 accuracy. Other techniques including batch normalization and regularization also have positive influence on the performance of the model.

5 Lessons Learned

Before including convolution layers, we first tried using only MaxPooling layers (without any convolution layers). As discussed in Section 4, this did not improve model performance (in fact, this seemed to hinder performance a bit). From this, we observe that MaxPooling layers are often best paired with convolution layers, and that using MaxPooling layers without convolution layers may not be the best architecture. Perhaps we experienced this because using MaxPooling on a raw image (without a convolutional layer) simply results in information loss, whereas using MaxPooling in tandem with convolution permits the filters in the convolution layer to learn to recognize features so that MaxPooling will capture presence of the feature (as indicated by the convolution layer) in a window's locality, passing this information about the presence of the feature in a corresponding window on to the next layer. This seems reasonable because when a pattern in the multi-channel image is similar to a filter in a convolution layer, the resulting value from the dot-product-like computation will be large, and therefore be selected from its window and passed on by the subsequent MaxPooling layer.

For early stopping, the patience parameter should be proportional to the epoch number at around 2% to 5%. For batch size, it mainly influences the time it consumes to train the model. The accuracy of the model, as found in our investigation, has little to do with this parameter.

When tuning the hyperparameters, we tried several techniques that we didn't go into details during lectures. As for an image classification problem, we learnt how convolution layers can improve the performance of the model significantly together with pooling layers. These layers can be viewed as a way of feature extraction in that they can be trained to learn the internal structure of the images which leads to the model's better understanding of the training examples. Another important technique is the batch normalization layer we added in between convolution layers and pooling layers. As we discussed during lectures, it's always preferred that the data points are centered at the origin and normalized because this makes the model faster and more stable. The theoretical detail is beyond the scope of our discussion here, but we have seen how batch normalization improved our model.

Project 2 Code Base

April 22, 2022

[26]:
```
import numpy as np                    # advanced math library

import tensorflow as tf
from tensorflow import keras
import pandas as pd

from keras.models import Sequential  # Model type to be used

from keras.layers.core import Dense, Dropout, Activation # Types of layers to␣
 ↪be used in our model
from keras.layers import Conv2D
from keras.layers import BatchNormalization
from keras import regularizers
from keras.utils import np_utils                         # NumPy related tools
from keras.callbacks import EarlyStopping
from keras.layers.core.flatten import Flatten
from keras.layers.pooling import MaxPool2D
```

1 Load Data

[ ]:
```
X_train = np.load('training_images.npy')
y_train = np.load('training_labels.npy')
```

[ ]:
```
print("X_train shape", X_train.shape)
print("y_train shape", y_train.shape)
```

```
X_train shape (60000, 28, 28)
y_train shape (60000,)
```

2 Process Input Vector

[ ]:
```
X_train = X_train.reshape(60000, 28, 28, 1)
# X_train = X_train.reshape(60000, 784)

X_train = X_train.astype('float32')
```

```
X_train /= 255.0
print(X_train.shape)
```

```
(60000, 28, 28, 1)
```

[ ]:
```
nb_classes = 10

Y_train = np_utils.to_categorical(y_train, nb_classes)
```

3 Build the model

[ ]:
```
callBack = EarlyStopping(monitor='val_accuracy', patience=10,␣
↪restore_best_weights=True)
```

[28]:
```
model = Sequential()
#convolution layers and pooling layers
model.add(Conv2D(filters=32, padding='same', kernel_size=(3,3),␣
↪activation='relu', input_shape=(28,28,1), kernel_regularizer=regularizers.
↪L2(1e-4)))
model.add(BatchNormalization()),
model.add(MaxPool2D((3,3)))
model.add(BatchNormalization()),
model.add(Conv2D(filters=64, padding='same', kernel_size=(4,4),␣
↪activation='relu', kernel_regularizer=regularizers.L2(1e-4)))
model.add(BatchNormalization()),
model.add(MaxPool2D((3,3)))
model.add(BatchNormalization()),

model.add(Flatten())
model.add(Dense(1024, activation='relu', kernel_regularizer=regularizers.
↪L2(1e-4)))
model.add(Dropout(0.2))
model.add(Dense(nb_classes, activation='softmax'))
```

[29]:
```
model.summary()
```

```
Model: "sequential_1"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d_2 (Conv2D)           (None, 28, 28, 32)        320

 batch_normalization_4 (Batc  (None, 28, 28, 32)       128
 hNormalization)

 max_pooling2d_2 (MaxPooling  (None, 9, 9, 32)         0
```

```
2D)

batch_normalization_5 (Batc  (None, 9, 9, 32)         128
hNormalization)

conv2d_3 (Conv2D)            (None, 9, 9, 64)         32832

batch_normalization_6 (Batc  (None, 9, 9, 64)         256
hNormalization)

max_pooling2d_3 (MaxPooling  (None, 3, 3, 64)         0
2D)

batch_normalization_7 (Batc  (None, 3, 3, 64)         256
hNormalization)

flatten_1 (Flatten)          (None, 576)              0

dense_1 (Dense)              (None, 1024)             590848

dropout_1 (Dropout)          (None, 1024)             0

dense_2 (Dense)              (None, 10)               10250

=================================================================
Total params: 635,018
Trainable params: 634,634
Non-trainable params: 384
_________________________________________________________________
```

[ ]:
```
model.compile(loss='categorical_crossentropy', optimizer='adam',␣
↪metrics=['accuracy'])
```

[ ]:
```
model.fit(X_train, Y_train, batch_size=32, epochs=60, validation_split=0.2,␣
↪verbose=1, callbacks=callBack)
```

```
Epoch 1/60
1500/1500 [==============================] - 98s 64ms/step - loss: 0.4774 -
accuracy: 0.8566 - val_loss: 0.4081 - val_accuracy: 0.8805
Epoch 2/60
1500/1500 [==============================] - 95s 63ms/step - loss: 0.3588 -
accuracy: 0.8993 - val_loss: 0.3988 - val_accuracy: 0.8817
Epoch 3/60
1500/1500 [==============================] - 93s 62ms/step - loss: 0.3230 -
accuracy: 0.9114 - val_loss: 0.3500 - val_accuracy: 0.9039
Epoch 4/60
1500/1500 [==============================] - 90s 60ms/step - loss: 0.3020 -
accuracy: 0.9201 - val_loss: 0.3381 - val_accuracy: 0.9123
```

```
Epoch 5/60
1500/1500 [==============================] - 91s 61ms/step - loss: 0.2887 -
accuracy: 0.9263 - val_loss: 0.3575 - val_accuracy: 0.9045
Epoch 6/60
1500/1500 [==============================] - 89s 60ms/step - loss: 0.2714 -
accuracy: 0.9322 - val_loss: 0.3512 - val_accuracy: 0.9054
Epoch 7/60
1500/1500 [==============================] - 90s 60ms/step - loss: 0.2596 -
accuracy: 0.9376 - val_loss: 0.3971 - val_accuracy: 0.8969
Epoch 8/60
1500/1500 [==============================] - 90s 60ms/step - loss: 0.2506 -
accuracy: 0.9416 - val_loss: 0.3728 - val_accuracy: 0.9048
Epoch 9/60
1500/1500 [==============================] - 91s 61ms/step - loss: 0.2388 -
accuracy: 0.9467 - val_loss: 0.3538 - val_accuracy: 0.9121
Epoch 10/60
1500/1500 [==============================] - 91s 60ms/step - loss: 0.2356 -
accuracy: 0.9479 - val_loss: 0.3517 - val_accuracy: 0.9173
Epoch 11/60
1500/1500 [==============================] - 90s 60ms/step - loss: 0.2288 -
accuracy: 0.9503 - val_loss: 0.3512 - val_accuracy: 0.9159
Epoch 12/60
1500/1500 [==============================] - 92s 61ms/step - loss: 0.2201 -
accuracy: 0.9535 - val_loss: 0.4024 - val_accuracy: 0.9009
Epoch 13/60
1500/1500 [==============================] - 93s 62ms/step - loss: 0.2146 -
accuracy: 0.9568 - val_loss: 0.3763 - val_accuracy: 0.9181
Epoch 14/60
1500/1500 [==============================] - 92s 61ms/step - loss: 0.2122 -
accuracy: 0.9581 - val_loss: 0.4175 - val_accuracy: 0.9044
Epoch 15/60
1500/1500 [==============================] - 90s 60ms/step - loss: 0.2068 -
accuracy: 0.9601 - val_loss: 0.3748 - val_accuracy: 0.9170
Epoch 16/60
1500/1500 [==============================] - 90s 60ms/step - loss: 0.2048 -
accuracy: 0.9614 - val_loss: 0.3806 - val_accuracy: 0.9163
Epoch 17/60
1500/1500 [==============================] - 93s 62ms/step - loss: 0.2020 -
accuracy: 0.9627 - val_loss: 0.4114 - val_accuracy: 0.9133
Epoch 18/60
1500/1500 [==============================] - 91s 61ms/step - loss: 0.2017 -
accuracy: 0.9634 - val_loss: 0.3785 - val_accuracy: 0.9172
Epoch 19/60
1500/1500 [==============================] - 92s 61ms/step - loss: 0.1944 -
accuracy: 0.9658 - val_loss: 0.4245 - val_accuracy: 0.9050
Epoch 20/60
1500/1500 [==============================] - 92s 61ms/step - loss: 0.1947 -
accuracy: 0.9663 - val_loss: 0.4104 - val_accuracy: 0.9117
```

```
Epoch 21/60
1500/1500 [==============================] - 91s 61ms/step - loss: 0.1933 -
accuracy: 0.9672 - val_loss: 0.4034 - val_accuracy: 0.9097
Epoch 22/60
1500/1500 [==============================] - 92s 62ms/step - loss: 0.1947 -
accuracy: 0.9667 - val_loss: 0.4074 - val_accuracy: 0.9139
Epoch 23/60
1500/1500 [==============================] - 92s 62ms/step - loss: 0.1888 -
accuracy: 0.9690 - val_loss: 0.4073 - val_accuracy: 0.9181
```

[ ]:
```
<keras.callbacks.History at 0x7f4140f5a790>
```

[ ]:
```
X_out = np.load('test_images.npy')
# X_out = X_out.reshape(10000, 784)   # reshape 10,000 28 x 28 matrices into␣
 ↪10,000 784-length vectors.
X_out = X_out.reshape(10000, 28, 28, 1)
X_out = X_out.astype('float32')   # change integers to 32-bit floating point␣
 ↪numbers
X_out /= 255.0
predicted_classes_probabilities = model.predict(X_out)
predicted_classes = np.argmax(predicted_classes_probabilities,axis=1)
df={'LABEL':predicted_classes}
df=pd.DataFrame(data=df)
df['LABEL'].to_csv('output.csv',index_label='ID')
```