Homework Assignment 5

CSE 251A: Introduction to Machine Learning

Due: June 8nd, 2022, 9:30am (Pacific Time)

Instructions: Please answer the questions below, attach your code in the document, and insert figures to create **a single PDF file**.

Grade: ____ out of 120 points

1 Activation functions (40 Points)

1. Which one of these is a valid layer? For this question, `w.shape=(input, output)` with `x.shape=(batch siz` and `b.shape=(1, output)`. Note, below that anywhere we use dot, we could have instead used matmul.

   (a) `z = activationFunction(np.dot(x, b) + w)`

   (b) `z = activationFunction(np.dot(x, w)) + b`

   [✓ (c)] `z = activationFunction(np.dot(x, w) + b)`

2. Name at least two possible activation functions and explain the reason why they are used as activation functions.

   Sigmoid function: An S-shaped curved graph with limits usually set between -1 to 1 or 0 to 1. It is commonly used in the layers of a neural network and can be used to solve non-linear datasets. Real values can also be read easily as probabilities when using this function as well. This can also be used in binary non-linear classification problems where values converge to either ends of the sigmoid function graph. They converge to either limit much quicker due to the S-shape of the graph when compared to linear activation functions.

   ReLu function: Is a piecewise linear function that will output the max of a function if the value is greater than zero, otherwise the output is 0. It has good performance and can be used on a larger array of datasets. It also overcomes the vanishing gradient problem and allows models to learn faster. It is also able to return a true zero output rather than sigmoid functions that approximate a zero value. This activation function can also introduce non-linearity to the patterns in the data it reads as well.

3. What will happen if the activation function is a linear function in Multi-layer Perceptron?

   The model would converge all on the same linear function since multiple linear functions are still linear functions. Without a non-linear activation function, the multilayered perceptron won't be able to learn non-linear relationships. The ability to learn complex relationships would be decreased since all the linear functions can be transformed and reduced down to one single linear function.

2 Overfitting and Regularization (30 Points)

1. What are the common techniques to alleviate overfitting in the neural network training?

   We can use dropout regularization technique to set a fraction of the inputs to 0 at each training step. This helps in reducing over-reliance on certain trains and adds randomibility to the dataset. You can also use data augmentation to apply random transformations to the data to increase the diversity of the training. You can also apply early stopping to stop the training when the validation performance starts to decline. We can also reduce the number of layers, neurons and use dimensionality reduction to prevent overfitting.

2. Can we still apply the L1/L2 regularization in NN? If we can, how; if we cannot, why?

   L1 regularization adds a penalty term to the loss function by using absolute value of the weights' sum. This allows the weights to be able to be zero which simplifies the model and can help increase performance and reduce overfitting. In particular it can be used as a neuron selection process because it can zero weights for hidden layer neurons.

   L2 regularization is the squared norm of the weights of each network's layer. this causes weights to decrease in value which leads to a more simplified model. L2 brings values towards to but not exactly zero. This method also prevents overfitting and simplifies the model and increases performance.

3 Compute output for a Convolutional Neural Network (30 Points)

Consider the image X and filter F given below. Let X be convolved with F using no padding and a stride of 1 to produce an output Y . What is the output Y ?

$$X = \begin{bmatrix} 1 & 0 & -2 & 3 & 4 & 1 \\ 2 & 9 & 5 & 6 & 0 & -1 \\ 0 & -3 & 1 & 3 & 4 & 4 \\ 6 & 5 & 2 & 0 & 6 & 8 \\ -5 & 4 & -3 & 1 & 3 & -2 \\ 4 & 1 & 2 & 8 & 9 & 7 \end{bmatrix}$$

$$F = \begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$$

Y[0, 0] = (-1 * 1) + (-1 * 0) + (-1 * -2) + (-1 * 2) + (8 * 9) + (-1 * 5) + (-1 * 0) + (-1 * -3) + (-1 * 1) = 41

Y[0, 1] = (-1 * 0) + (-1 * -2) + (-1 * 3) + (-1 * 9) + (8 * 5) + (-1 * 6) + (-1 * -3) + (-1 * 1) + (-1 * 3) = 34

Y[0, 2] = (-1 * -2) + (-1 * 3) + (-1 * 4) + (-1 * 5) + (8 * 0) + (-1 * -1) + (-1 * 1) + (-1 * 3) + (-1 * 4) = 43

Y[0, 3] = (-1 * 3) + (-1 * 4) + (-1 * 1) + (-1 * 0) + (8 * 6) + (-1 * 3) + (-1 * 4) + (-1 * 4) + (-1 * 6) = 16

Y[1, 0] = (-1 * 2) + (-1 * 9) + (-1 * 5) + (-1 * -5) + (8 * 4) + (-1 * 1) + (-1 * 0) + (-1 * -2) + (-1 * 3) = 5

Y[1, 1] = (-1 * 9) + (-1 * 5) + (-1 * 6) + (-1 * 4) + (8 * 0) + (-1 * 8) + (-1 * -2) + (-1 * 3) + (-1 * 4) = 35

Y[1, 2] = (-1 * 5) + (-1 * 6) + (-1 * 0) + (-1 * -2) + (8 * 6) + (-1 * 2) + (-1 * 3) + (-1 * 4) + (-1 * 6) = 43

Y[1, 3] = (-1 * 6) + (-1 * 0) + (-1 * 6) + (-1 * 4) + (8 * 4) + (-1 * 3) + (-1 * 4) + (-1 * 6) + (-1 * 8) = 12

Y[2, 0] = (-1 * -5) + (-1 * 4) + (-1 * -3) + (-1 * 1) + (8 * 1) + (-1 * 2) + (-1 * 2) + (-1 * 8) + (-1 * 9) = 6

Y[2, 1] = (-1 * 4) + (-1 * -3) + (-1 * 1) + (-1 * 3) + (8 * 3) + (-1 * 8) + (-1 * 8) + (-1 * 9) + (-1 * 7) = 17

Y[2, 2] = (-1 * -3) + (-1 * 1) + (-1 * 3) + (-1 * 1) + (8 * 6) + (-1 * 9) + (-1 * 9) + (-1 * 7) + (-1 * 4) = 27

Y[2, 3] = (-1 * 1) + (-1 * 3) + (-1 * 4) + (-1 * 2) + (8 * 9) + (-1 * 7) + (-1 * 7) + (-1 * 4) + (-1 * 1) = 32

Y[3, 0] = (-1 * 6) + (-1 * 5) + (-1 * 4) + (-1 * -5) + (8 * 4) + (-1 * 3) + (-1 * 1) + (-1 * -2) + (-1 * 2) = 12

Y[3, 1] = (-1 * 5) + (-1 * 4) + (-1 * 6) + (-1 * 4) + (8 * 0) + (-1 * 4) + (-1 * -2) + (-1 * 2) + (-1 * 3) = 17

Y[3, 2] = (-1 * 4) + (-1 * 6) + (-1 * 0) + (-1 * -2) + (8 * 6) + (-1 * 9) + (-1 * 2) + (-1 * 3) + (-1 * 4) = 36

Y[3, 3] = (-1 * 6) + (-1 * 0) + (-1 * 6) + (-1 * 4) + (8 * 4) + (-1 * 6) + (-1 * 4) + (-1 * 4) + (-1 * 6) = 30

$$Y = \begin{bmatrix} 41 & 34 & 43 & 16 \\ 5 & 35 & 43 & 12 \\ 6 & 17 & 27 & 32 \\ 12 & 17 & 36 & 30 \end{bmatrix}$$

4 (Bonus, 20 points) Experiment with CNN using Keras

In this question, you will experiment with Convolutional Neural Networks using the deep learning framework Keras (https://keras.io/api/). Please download the Jupyter notebook `HW5_CNN.ipynb` and fill in the blanks and answer the questions. Please attach your **code** and **answers** in Gradescope submission.

Note: Make sure this notebook is launched in an environment with Numpy, Tensorflow, matplotlib and Keras installed. You can refer to: https://www.tutorialspoint.com/keras/keras_installation.htm if you need help with creating a virtual environment with all required dependencies.

HW5: Image classification with Convolutional Neural Networks (20 points)

For this assignment, you'll build simple convolutional neural networks using Keras for image classification tasks. The goal is to get you familiar with the steps of working with deep learning models, namely, preprocessing dataset, defining models, train/test models and quantatively comparing performances. Make sure this notebook is launched in an environment with Numpy, Tensorflow, matplotlib and Keras installed. Refer to: https://www.tutorialspoint.com/keras/keras_installation.htm if you need help with creating a virtual environment with all required dependencies.

Furthermore, you can refer to the official Keras website for detailed documentations about different neural network layers (https://keras.io/api/layers/) and other classes.

In [3]:
```
from keras.datasets import mnist
import matplotlib.pyplot as plt
from keras.utils import np_utils
from keras.models import Sequential
from keras.layers import Dense, Dropout, Conv2D, MaxPool2D, Flatten
from keras.optimizers import SGD
import numpy as np
```

(1) Sample code (5 points)

As in class, we first download the MNIST dataset and get the train/test sets. We then process the data to be ready for training and testing.

In [4]:
```
# loading the dataset
(trainX, trainY), (testX, testY) = mnist.load_data()
```

In [5]:
```
def process_dataset(trainX, trainY, testX, testY):
    # reshape features and normalize
    trainX = trainX.reshape((trainX.shape[0], 28, 28, 1))
    testX = testX.reshape((testX.shape[0], 28, 28, 1))
    trainX = trainX.astype('float32')
    testX = testX.astype('float32')
    trainX = trainX / 255.0
    testX = testX / 255.0
    # converting labels to one-hot encoding
    trainY = np_utils.to_categorical(trainY)
    testY = np_utils.to_categorical(testY)
    return trainX, trainY, testX, testY
trainX, trainY, testX, testY = process_dataset(trainX, trainY, testX, testY)
```

We then define the model. Similar to in-class demo, this model has 1 convolution layer with 32 filters, followed by one 2-by-2 MaxPooling layer. The output from MaxPooling layer is then flattened and goes through two linear layers, with 100 and 10 hidden units respectively. We use Stochastic Gradient Descent as our optimizer, and we can adjust its learning rate.

In [6]:
```
def define_model(learning_rate):
    model = Sequential()
    model.add(Conv2D(32, kernel_size=(3,3), strides=(1,1), padding='valid', activation='relu', input_shape=(28,28,1)))
    model.add(MaxPool2D((2, 2)))
    model.add(Flatten())
    model.add(Dense(100, activation='relu'))
    model.add(Dense(10, activation='softmax'))
    # compile model
    opt = SGD(lr=learning_rate)
    model.compile(optimizer=opt, loss='categorical_crossentropy', metrics=['accuracy'])
    return model
```

Now we can train and evaulate the specified model. Here we're using the test set as the validation set for simplicity. However, to be more rigorous we often split the training dataset into train/validation sets and tune the hyperparameters using only the training dataset, and we test the model on the test set after figuring out the best hyperparameters.

In [7]:
```
# here we define a model with lr=0.01
model = define_model(0.01)
history = model.fit(trainX, trainY, batch_size=32, epochs=10, validation_data=(testX, testY))
```

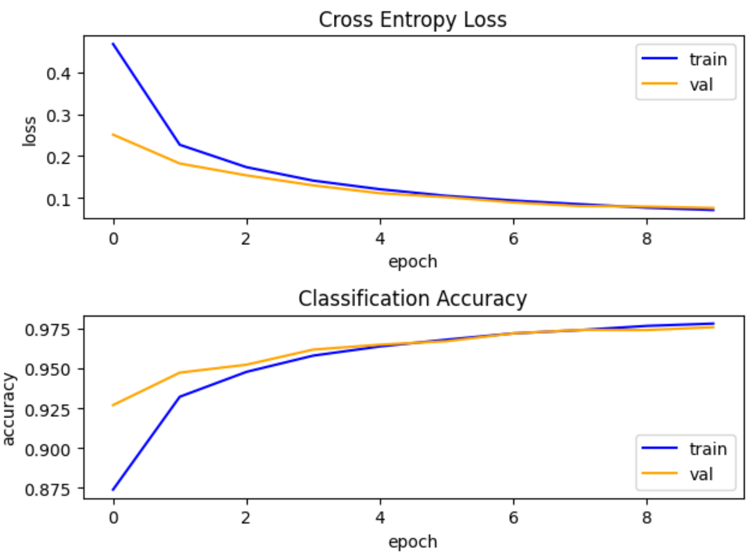
```
Epoch 1/10
C:\Users\mwood\AppData\Local\Programs\Python\Python310\lib\site-packages\keras\optimizers\legacy\gradient_descent.py:114: UserWarning: The `lr` argume
nt is deprecated, use `learning_rate` instead.
  super().__init__(name, **kwargs)
```

```
1875/1875 [==============================] - 10s 5ms/step - loss: 0.4673 - accuracy: 0.8737 - val_loss: 0.2512 - val_accuracy: 0.9269
Epoch 2/10
1875/1875 [==============================] - 10s 5ms/step - loss: 0.2270 - accuracy: 0.9321 - val_loss: 0.1823 - val_accuracy: 0.9473
Epoch 3/10
1875/1875 [==============================] - 10s 5ms/step - loss: 0.1736 - accuracy: 0.9478 - val_loss: 0.1541 - val_accuracy: 0.9523
Epoch 4/10
1875/1875 [==============================] - 10s 6ms/step - loss: 0.1416 - accuracy: 0.9580 - val_loss: 0.1303 - val_accuracy: 0.9618
Epoch 5/10
1875/1875 [==============================] - 10s 5ms/step - loss: 0.1210 - accuracy: 0.9638 - val_loss: 0.1115 - val_accuracy: 0.9649
Epoch 6/10
1875/1875 [==============================] - 11s 6ms/step - loss: 0.1053 - accuracy: 0.9681 - val_loss: 0.1021 - val_accuracy: 0.9671
Epoch 7/10
1875/1875 [==============================] - 11s 6ms/step - loss: 0.0942 - accuracy: 0.9720 - val_loss: 0.0896 - val_accuracy: 0.9720
Epoch 8/10
1875/1875 [==============================] - 11s 6ms/step - loss: 0.0854 - accuracy: 0.9741 - val_loss: 0.0809 - val_accuracy: 0.9742
Epoch 9/10
1875/1875 [==============================] - 11s 6ms/step - loss: 0.0769 - accuracy: 0.9767 - val_loss: 0.0796 - val_accuracy: 0.9741
Epoch 10/10
1875/1875 [==============================] - 11s 6ms/step - loss: 0.0714 - accuracy: 0.9782 - val_loss: 0.0765 - val_accuracy: 0.9759
```

Once training is completed, we can plot the train/validation losses and train/validation accuracies.

In [8]:
```python
#plot loss
fig = plt.figure()
plt.subplot(2, 1, 1)
plt.title('Cross Entropy Loss')
plt.plot(history.history['loss'], color='blue', label='train')
plt.plot(history.history['val_loss'], color='orange', label='val')
plt.legend(('train','val'))
plt.xlabel('epoch')
plt.ylabel('loss')

# plot accuracy
plt.subplot(2, 1, 2)
plt.title('Classification Accuracy')
plt.plot(history.history['accuracy'], color='blue', label='train')
plt.plot(history.history['val_accuracy'], color='orange', label='test')
plt.legend(('train','val'))
plt.xlabel('epoch')
plt.ylabel('accuracy')
fig.tight_layout()
plt.show()
```



Question 1 (5 points):

What do you observe in the above plots? What do you think might be the reason?

Your Answer

That the training set has a higher initial loss but sees a sharper decrease when compared to the validation set. Since we are using the test set for validation, we see the model starts at a smaller loss but converges at the same point as the training set. To better test this model, we should use a seperate dataset that isn't associated with the test dataset.

(2) Vary learning rates (5 points)

Recall from lecture that we update the weights of the neural network by first calculate the gradients with backpropagation from the loss $L$, then update the weights by $$ w = w - \eta*\frac{\partial L}{\partial w}$$ Here, $\eta$ is the learning rate and decides the step size of updates. Previously we used $\eta=0.01$. We want to see the effect of learning rate on the training process, therefore we would like to try two other choices of $\eta$. (1) $\eta=1$ (2) $\eta=$1e-5 (0.00001)

```
In [10]: #### TODO 1 STARTS ###
         model_eta_large = define_model(1)
         history_eta_large = model_eta_large.fit(trainX, trainY, batch_size=32, epochs=10, validation_data=(testX, testY))
         #### TODO 1 ENDS ###
```

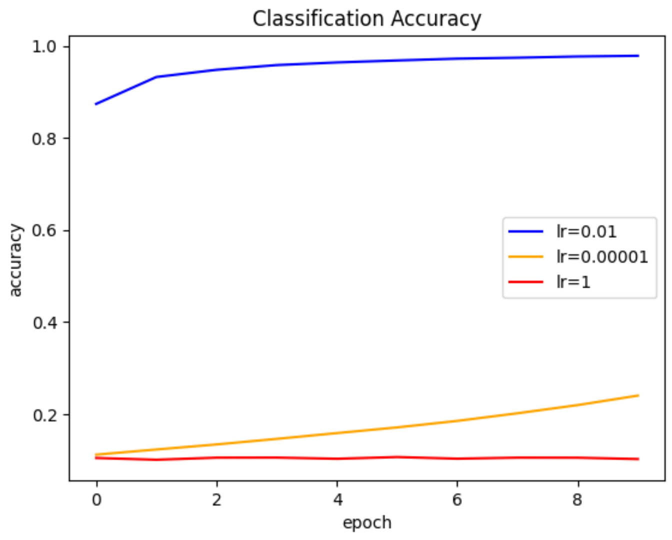
```
Epoch 1/10
1875/1875 [==============================] - 10s 5ms/step - loss: 2.3124 - accuracy: 0.1044 - val_loss: 2.3051 - val_accuracy: 0.0982
Epoch 2/10
1875/1875 [==============================] - 10s 5ms/step - loss: 2.3091 - accuracy: 0.1005 - val_loss: 2.3067 - val_accuracy: 0.0974
Epoch 3/10
1875/1875 [==============================] - 10s 5ms/step - loss: 2.3083 - accuracy: 0.1051 - val_loss: 2.3053 - val_accuracy: 0.0974
Epoch 4/10
1875/1875 [==============================] - 10s 5ms/step - loss: 2.3086 - accuracy: 0.1052 - val_loss: 2.3050 - val_accuracy: 0.1135
Epoch 5/10
1875/1875 [==============================] - 10s 5ms/step - loss: 2.3088 - accuracy: 0.1027 - val_loss: 2.3098 - val_accuracy: 0.1010
Epoch 6/10
1875/1875 [==============================] - 10s 5ms/step - loss: 2.3086 - accuracy: 0.1064 - val_loss: 2.3121 - val_accuracy: 0.1135
Epoch 7/10
1875/1875 [==============================] - 10s 5ms/step - loss: 2.3091 - accuracy: 0.1028 - val_loss: 2.3034 - val_accuracy: 0.0958
Epoch 8/10
1875/1875 [==============================] - 10s 5ms/step - loss: 2.3083 - accuracy: 0.1051 - val_loss: 2.3065 - val_accuracy: 0.1010
Epoch 9/10
1875/1875 [==============================] - 10s 5ms/step - loss: 2.3085 - accuracy: 0.1050 - val_loss: 2.3058 - val_accuracy: 0.1028
Epoch 10/10
1875/1875 [==============================] - 10s 5ms/step - loss: 2.3087 - accuracy: 0.1020 - val_loss: 2.3141 - val_accuracy: 0.1135
```

```
In [11]: #### TODO 2 STARTS ###
         model_eta_small = define_model(0.00001)
         history_eta_small = model_eta_small.fit(trainX, trainY, batch_size=32, epochs=10, validation_data=(testX, testY))
         #### TODO 2 ENDS ###
```

```
Epoch 1/10
1875/1875 [==============================] - 10s 5ms/step - loss: 2.3056 - accuracy: 0.1117 - val_loss: 2.3023 - val_accuracy: 0.1184
Epoch 2/10
1875/1875 [==============================] - 10s 5ms/step - loss: 2.2996 - accuracy: 0.1228 - val_loss: 2.2963 - val_accuracy: 0.1299
Epoch 3/10
1875/1875 [==============================] - 10s 5ms/step - loss: 2.2937 - accuracy: 0.1338 - val_loss: 2.2903 - val_accuracy: 0.1434
Epoch 4/10
1875/1875 [==============================] - 10s 5ms/step - loss: 2.2879 - accuracy: 0.1459 - val_loss: 2.2844 - val_accuracy: 0.1562
Epoch 5/10
1875/1875 [==============================] - 10s 5ms/step - loss: 2.2821 - accuracy: 0.1585 - val_loss: 2.2785 - val_accuracy: 0.1677
Epoch 6/10
1875/1875 [==============================] - 10s 5ms/step - loss: 2.2763 - accuracy: 0.1708 - val_loss: 2.2726 - val_accuracy: 0.1814
Epoch 7/10
1875/1875 [==============================] - 10s 5ms/step - loss: 2.2705 - accuracy: 0.1849 - val_loss: 2.2666 - val_accuracy: 0.1971
Epoch 8/10
1875/1875 [==============================] - 10s 5ms/step - loss: 2.2646 - accuracy: 0.2015 - val_loss: 2.2606 - val_accuracy: 0.2138
Epoch 9/10
1875/1875 [==============================] - 10s 5ms/step - loss: 2.2587 - accuracy: 0.2193 - val_loss: 2.2546 - val_accuracy: 0.2335
Epoch 10/10
1875/1875 [==============================] - 10s 5ms/step - loss: 2.2527 - accuracy: 0.2399 - val_loss: 2.2484 - val_accuracy: 0.2539
```

We now compare the training accuracy of the two above models with the training accuracy of the model in part 1.

```
In [12]: plt.title('Classification Accuracy')
         plt.plot(history.history['accuracy'], color='blue')
         plt.plot(history_eta_small.history['accuracy'], color='orange')
         plt.plot(history_eta_large.history['accuracy'], color='red')
         plt.legend(('lr=0.01','lr=0.00001','lr=1'))
         plt.xlabel('epoch')
         plt.ylabel('accuracy')
         plt.show()
```



Question 2 (5 points):

What do you observe by looking at the training accuracies above? Does the two other models with small and large learning rates seem to be learning? What do you think might be the reason? (optional) Can you find a better learning rate than the baseline?

Your Answer

The large and small learning rates both have poor accuracy increase. A high learning rate can cause instability and divergence which causes the model to overshoot the optimal solution. In the case of a very small lerning rate, the models training is too slow and may not converge fast enough onto an optimal solution which leads to poor learning and suboptimal accuracy.

(3) Adding momentum (5 points)

Till now we have tried various learning rates with SGD. There are various ways to make SGD behave more intelligently, one of which is momentum. Intuitively, when SGD tries to descend down a valley (an analogy for the case where the gradient of one dimension is larger than gradient of another dimension), SGD might bounce between the walls of the valley instead of descending along the valley. This makes SGD converge slower or even stuck. Momentum works by dampening the oscillations of SGD and encourages it to follow a smoother path. Formally, SGD with momentum update weights by the following way:

$$z^{k+1} = \beta z^{k} + \frac{\partial L}{\partial w^k}$$$$w^{k+1} = w^{k} - \eta*z^{k+1}$$
Here $\beta$ is the momentum and is between 0 and 1. The official documentation of SGD details how to specify momentum (https://keras.io/api/optimizers/sgd/). If you want to learn more about momentum, this post might be helpful: https://distill.pub/2017/momentum/

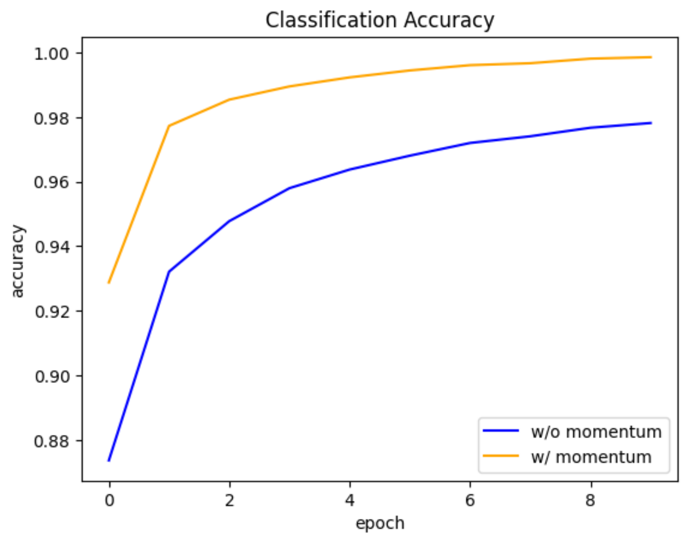
Please define a model with learning rate 0.01 and momentum 0.9, then compare it to the baseline in part 1.

In [13]:
```
def define_model_with_momentum(learning_rate,momentum):
    model = Sequential()
    model.add(Conv2D(32, kernel_size=(3,3), strides=(1,1), padding='valid', activation='relu', input_shape=(28,28,1)))
    model.add(MaxPool2D((2, 2)))
    model.add(Flatten())
    model.add(Dense(100, activation='relu'))
    model.add(Dense(10, activation='softmax'))
    # compile model
    #### TODO 3 STARTS ###
    opt = SGD(learning_rate=learning_rate, momentum=momentum)
    #### TODO 3 ENDS ###
    model.compile(optimizer=opt, loss='categorical_crossentropy', metrics=['accuracy'])
    return model
```

In [14]:
```
#### TODO 4 STARTS ###
model_momentum = define_model_with_momentum(0.01, 0.9)
history_momentum = model_momentum.fit(trainX, trainY, batch_size=32, epochs=10, validation_data=(testX, testY))
#### TODO 4 ENDS ###
```

```
Epoch 1/10
1875/1875 [==============================] - 11s 6ms/step - loss: 0.2355 - accuracy: 0.9288 - val_loss: 0.0966 - val_accuracy: 0.9715
Epoch 2/10
1875/1875 [==============================] - 10s 6ms/step - loss: 0.0753 - accuracy: 0.9773 - val_loss: 0.0721 - val_accuracy: 0.9759
Epoch 3/10
1875/1875 [==============================] - 10s 5ms/step - loss: 0.0476 - accuracy: 0.9854 - val_loss: 0.0527 - val_accuracy: 0.9822
Epoch 4/10
1875/1875 [==============================] - 10s 5ms/step - loss: 0.0336 - accuracy: 0.9895 - val_loss: 0.0443 - val_accuracy: 0.9868
Epoch 5/10
1875/1875 [==============================] - 10s 6ms/step - loss: 0.0246 - accuracy: 0.9923 - val_loss: 0.0494 - val_accuracy: 0.9838
Epoch 6/10
1875/1875 [==============================] - 10s 6ms/step - loss: 0.0179 - accuracy: 0.9945 - val_loss: 0.0402 - val_accuracy: 0.9867
Epoch 7/10
1875/1875 [==============================] - 10s 6ms/step - loss: 0.0133 - accuracy: 0.9961 - val_loss: 0.0418 - val_accuracy: 0.9870
Epoch 8/10
1875/1875 [==============================] - 11s 6ms/step - loss: 0.0105 - accuracy: 0.9967 - val_loss: 0.0427 - val_accuracy: 0.9867
Epoch 9/10
1875/1875 [==============================] - 11s 6ms/step - loss: 0.0074 - accuracy: 0.9981 - val_loss: 0.0466 - val_accuracy: 0.9873
Epoch 10/10
1875/1875 [==============================] - 11s 6ms/step - loss: 0.0055 - accuracy: 0.9986 - val_loss: 0.0423 - val_accuracy: 0.9881
```

In [15]:
```
plt.title('Classification Accuracy')
plt.plot(history.history['accuracy'], color='blue')
plt.plot(history_momentum.history['accuracy'], color='orange')
plt.legend(('w/o momentum','w/ momentum'))
plt.xlabel('epoch')
plt.ylabel('accuracy')
plt.show()
```



Question 3 (5 points):

What do you observe in the plot? Does momentum improves training?

Your Answer

That the model with momentum improves the accuraccy of the model. This is due to momentum enabling faster convergence of the model onto optimal solutions. This is done by accumulating velocity in the direction of consistent gradients. this is to account for flat regions in the gradient descent and converge on likely optimal solutions more easily. It also helps improve training by smoothing the optimization path as well by dampening the oscillation of SGD on a minimum. This allows updates to the parameters to be more consistent.

(4) Adding convolution layers (5 points)

To increase model capacity (the ability to fit more complex dataset), one way is to adding layers to the model. In part 1, the model given to you has the following layers before the final 2 dense layers:

(1) 2D convolution with 32 filters of size 3-by-3, stride 1-by-1, 'valid' padding and relu activations

(2) 2-by-2 Max Pooling layer

(2) Flatten layer

In the function below, please implement a model with the following layers (in this order):

(1) 2D convolution with 32 filters of size 3-by-3, stride 1-by-1, 'valid' padding and relu activations

(2) 2-by-2 Max Pooling layer

(1) 2D convolution with 64 filters of size 3-by-3, stride 1-by-1, 'valid' padding and relu activations

(2) 2-by-2 Max Pooling layer

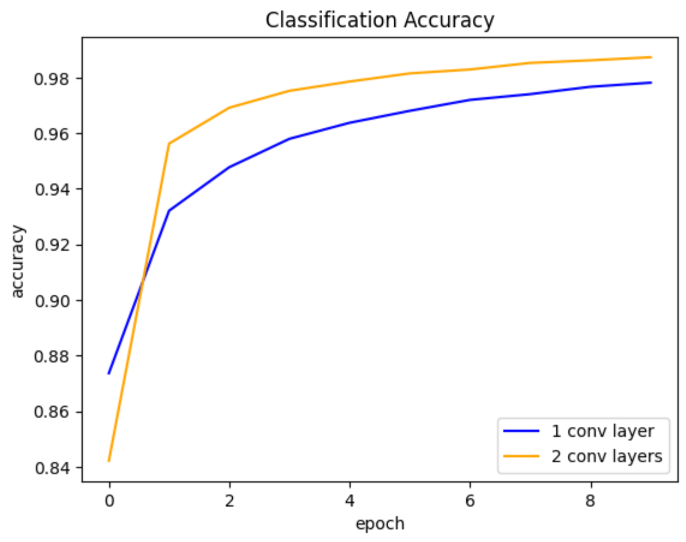
(2) Flatten layer

In [17]:
```
def define_model_2_conv(learning_rate):
    model = Sequential()
    #### TODO 5 STARTS ###
    # adding layers here
    model.add(Conv2D(32, kernel_size=(3,3), strides=(1,1), padding='valid', activation='relu', input_shape=(28,28,1)))
    model.add(MaxPool2D((2, 2)))
    model.add(Conv2D(64, kernel_size=(3,3), strides=(1,1), padding='valid', activation='relu'))
    model.add(MaxPool2D((2, 2)))
    model.add(Flatten())
    #### TODO 5 ENDS ###
    model.add(Dense(100, activation='relu'))
    model.add(Dense(10, activation='softmax'))
    # compile model
    opt = SGD(lr=learning_rate)
    model.compile(optimizer=opt, loss='categorical_crossentropy', metrics=['accuracy'])
    return model
```

In [18]:
```
# define model and train
#### TODO 6 STARTS ###
model_2_layer = define_model_2_conv(.01)
history_2_layer = model_2_layer.fit(trainX, trainY, batch_size=32, epochs=10, validation_data=(testX, testY))
#### TODO 6 ENDS ###
```

```
Epoch 1/10
1875/1875 [==============================] - 13s 7ms/step - loss: 0.5345 - accuracy: 0.8422 - val_loss: 0.1876 - val_accuracy: 0.9438
Epoch 2/10
1875/1875 [==============================] - 13s 7ms/step - loss: 0.1446 - accuracy: 0.9563 - val_loss: 0.0971 - val_accuracy: 0.9718
Epoch 3/10
1875/1875 [==============================] - 13s 7ms/step - loss: 0.1006 - accuracy: 0.9692 - val_loss: 0.0835 - val_accuracy: 0.9735
Epoch 4/10
1875/1875 [==============================] - 13s 7ms/step - loss: 0.0802 - accuracy: 0.9753 - val_loss: 0.0704 - val_accuracy: 0.9774
Epoch 5/10
1875/1875 [==============================] - 12s 7ms/step - loss: 0.0689 - accuracy: 0.9786 - val_loss: 0.0549 - val_accuracy: 0.9829
Epoch 6/10
1875/1875 [==============================] - 12s 7ms/step - loss: 0.0594 - accuracy: 0.9815 - val_loss: 0.0607 - val_accuracy: 0.9804
Epoch 7/10
1875/1875 [==============================] - 12s 7ms/step - loss: 0.0543 - accuracy: 0.9829 - val_loss: 0.0529 - val_accuracy: 0.9834
Epoch 8/10
1875/1875 [==============================] - 12s 7ms/step - loss: 0.0481 - accuracy: 0.9853 - val_loss: 0.0430 - val_accuracy: 0.9859
Epoch 9/10
1875/1875 [==============================] - 12s 7ms/step - loss: 0.0442 - accuracy: 0.9862 - val_loss: 0.0414 - val_accuracy: 0.9864
Epoch 10/10
1875/1875 [==============================] - 12s 7ms/step - loss: 0.0407 - accuracy: 0.9873 - val_loss: 0.0436 - val_accuracy: 0.9857
```

In [20]:
```
plt.title('Classification Accuracy')
plt.plot(history.history['accuracy'], color='blue')
plt.plot(history_2_layer.history['accuracy'], color='orange')
plt.legend(('1 conv layer','2 conv layers'))
plt.xlabel('epoch')
plt.ylabel('accuracy')
plt.show()
```



Question 4 (5 points):

What do you observe in the plot? Does adding a covolutional layer improves training set accuracy? What might be the reason to the improvement if there are any?

Your Answer

Adding more convolutional layers improves the model. Adding more layes can increase the ability of the model to capture relationships found within the data. It also allows for feature hierarchy to be better captured within the dataset and better detailing to learn high-level features. It is also more robust to outliers or invariant features foudn within the data. Overfitting can occur if the model becomes too complex, which can be caused by adding too many convolutional layers.

In [ ]:

5 (20 points) Convolution with padding and dilation.

In Lecture 15, slide 12 (and Q3 of this assignment) we have gone through the process of calculating the output size of a convolutional layer. Besides the height and width, dilation, stride, and padding are also important parameters of a CNN layer.

- dilation controls the spacing between the kernel points; also known as the atrous algorithm.
- stride controls the stride for the cross-correlation, a single number or a tuple.
- padding controls the amount of padding applied to the input's outer edges.

This link provides a great visualization of what each parameter controls.

1. Given an input of size (16, 16), a filter of size (2, 2) with `stride=2, padding=1, dilation=4`, what's the output size of it after a **one** layer 2d-convolution?

   output = (W - (F*D) + 2P)/S + 1 output = (16 - (2*4) + 2(1))/2 + 1 output = (16 - 8 + 2)/2 + 1 = 10/2 + 1 = 6

2. Given an input of size (32, 32), a filter of size (3, 3) with `stride=2, padding=1, dilation=1`, what's the output size of it after a **two** layer 2d-convolution?

   output = (32 - (3*1) + 2(1))/2 + 1

   output = (32 -3 + 2)/2 + 1 = (29 + 2)/2 + 1 = 31/2 + 1 = 16.5 = 16

   output = (16 - (3*1) + 2(1))/2 + 1

   output = (16 - 3 + 2)/2 + 1 = 15/2 + 1 = 8.5 = 8

3. (**Receptive Field**) The Receptive Field (RF) is defined as the size of the region in the input or previous layers that produce the current feature. For example, Figure 1 shows a 3-layer network of filter size (3, 3), with `stride=1, padding=2, dilation=1`. The receptive field in layer 2 for position (2, 2) of layer 3 is $3 \times 3 = 9$, namely, position (1, 1), (1, 2), (1, 3), (2, 1), (2, 2), (2, 3), (3, 1), (3, 2), (3, 3) of layer 2.

   When we want to figure out the RF in layer 1, we need to take the **union** of the receptive field, i.e. the 9 positions, in layer 2. **Question**: assumed the input is large enough, say, (1000, 1000), what's the size of the receptive field in layer 1 for the position (2, 2) of layer 3?

   RF Layer 1 = (W - (F*D) + 2P)/S + 1

   RF Layer 1 = (9 - (3*1) + 2(2))/1 + 1

   RF Layer 1 = (9 - 3 + 4)/1 + 1 = 10/1 + 1 = 11

   RF Layer 1 = (11,11)

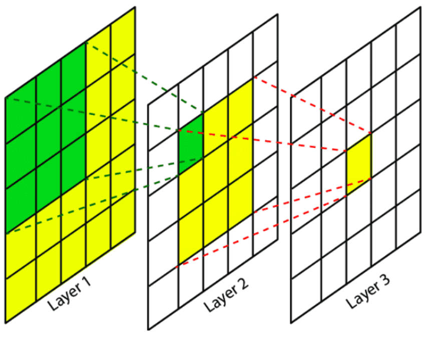


Figure 1: Example of RF for a 3-layer CNNs.