Policies

- Due 9 PM February 2nd 2018 via Moodle.
- You are free to collaborate on all the problems, subject to the collaboration policy stated in the syllabus.
- You should submit all code used in the homework. We ask that you use Python 3 and sklearn version 0.19 for your code, and that you comment your code such that the TAs can follow along and run it without any issues.
- This set requires the installation of both Tensorflow and Keras. There will be a recitation and office hour dedicated to helping you install these packages if you have problems.

Submission Instructions

Please submit your assignment as a .zip archive with filename `LastnameFirstname.zip` (replacing `Lastname` with your last name and `Firstname` with your first name), containing a PDF of your assignment writeup **in the main directory** with filename `LastnameFirstname_Set4.pdf` and your code files **in a directory named `LastnameFirstname`**. Failure to do so will result in a **2 point deduction**. Submit your code as Jupyter notebook .ipynb files or .py files, and **include any images generated by your code along with your answers in the solution .pdf file.**

1 Deep Learning Principles [35 Points]

Relevant materials: lectures on deep learning

Problem A [10 points]: Backpropagation and Weight Initialization

For this problem, we'll be utilizing the Tensorflow Playground to visualize/fit a neural network.

i. [5 points]: Fit the neural network at this link for about 250 iterations, and then do the same for the neural network at this link. Both networks have the same architecture and use ReLU activations. The only difference between the two is how the layer weights were initialized – you can examine the layer weights by hovering over the edges between neurons.

Give a mathematical justification, based on what you know about the backpropagation algorithm and the ReLU function, for the difference in the performance of the two networks.

Solution A.i:

During backpropagation, the gradient $\frac{\delta \mathcal{L}}{\delta W^\ell}$ is proportional to $\frac{\delta S^{\ell+1}}{\delta X^\ell} = W^{\ell+1}$ for hidden layers $\ell = 1, 2, ...L-1$ (all layers besides the output layer). Thus, if the weights of a neural network are initialized to zero, the initial gradients for all weights besides those entering the output layer will also be zero.

We now analyze the gradients for the weights W^L entering the output layer. Note that the ReLU activation function outputs 0 when its input is 0; we have $\text{ReLU}(0) = \max(0, 0) = 0$. We also know that $\frac{\delta \mathcal{L}}{\delta W^L}$ is proportional to $\frac{\delta S^L}{\delta W^L} = X^{L-1}$. When all the weights are initialized to 0, the input signal S^ℓ to the ReLU units is 0, so $X^{\ell-1} = 0$ for $\ell = 1, 2, ...L$. Thus, the gradient for W^L is also zero ($\frac{\delta \mathcal{L}}{\delta W^L} = 0$). Since all weights in the network have a gradient of zero, learning via gradient descent is not possible.

ii. [5 points]: Reset the two demos from part i (there is a reset button to the left of the "Run" button), change the activation functions of the neurons to sigmoid instead of ReLU, and train each of them for 4000 iterations.

Explain the differences in the models learned, and the speed at which they were learned, from those of part i in terms of the backpropagation algorithm and the sigmoid function.

Solution A.ii: *Sigmoid units have non-zero output when their input is zero, unlike ReLU units. Thus, $X^{L-1} \neq 0$ during the first iteration of gradient descent, allowing for a non-zero initial update to W^L. This allows for non-zero updates in preceding layers of the network during successive iterations of gradient descent, enabling learning.*

Using non-zero initial weights speeds convergence. We know $\frac{\delta \mathcal{L}}{\delta W^\ell}$ is proportional to $W^{\ell+1}$ for all layers besides the output layer. When the weights W^ℓ are initialized to zero, the multiplicative effect of this relationship makes initial updates to the network's first layers very small. Initializing with non-zero weights allows for larger *updates in the network's initial layers early on during training, hastening convergence.*

Problem B: [10 Points]

When training any model using SGD, it's important to shuffle your data to avoid correlated samples. To illustrate one reason for this that is particularly important for ReLU networks, consider a dataset of 1000 points, 500 of which have positive (+1) labels, and 500 of which have negative (-1) labels. What happens if we train a fully-connected network with ReLU activations using SGD, looping through all the negative examples before any of the positive examples? (Hint: this is called the "dying ReLU" problem.)

Solution B: *The ReLU units "die" – their incoming weights are updated towards zero and become negative. For $S < 0$, the gradient of the rectifier function ReLU$(S) = \max(0, S)$ is 0, so the weights entering the ReLU units stop changing and the units simply output 0.*

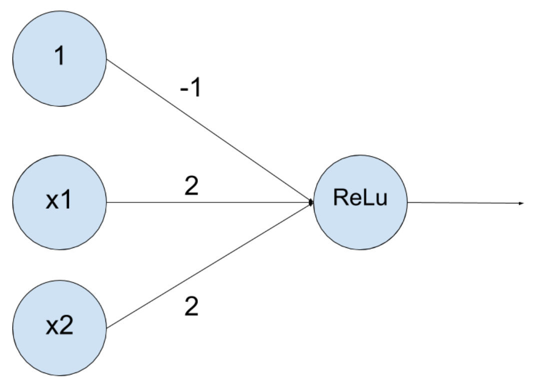
Problem C: Approximating Functions **[15 Points]**

i. [7 points]: Draw or describe a fully-connected network with ReLU units that implements the OR function on two 0/1-valued inputs, x_1 and x_2. Your networks should contain the minimum number of hidden units possible. The OR function OR(x_1, x_2) is defined as:

$$\text{OR}(1,0) \geq 1$$
$$\text{OR}(0,1) \geq 1$$
$$\text{OR}(1,1) \geq 1$$
$$\text{OR}(0,0) = 0$$

Your network need only produce the correct output when $x_1 \in \{0, 1\}$ and $x_2 \in \{0, 1\}$ (as described in the examples above).

Solution C.i: *A network consisting of a single hidden layer can approximate the OR function, as shown below:*



ii. [8 points]: What is the minimum number of fully-connected layers (with ReLU units) needed to implement an XOR of two 0/1-valued inputs x_1, x_2? Recall that the XOR function is defined as:

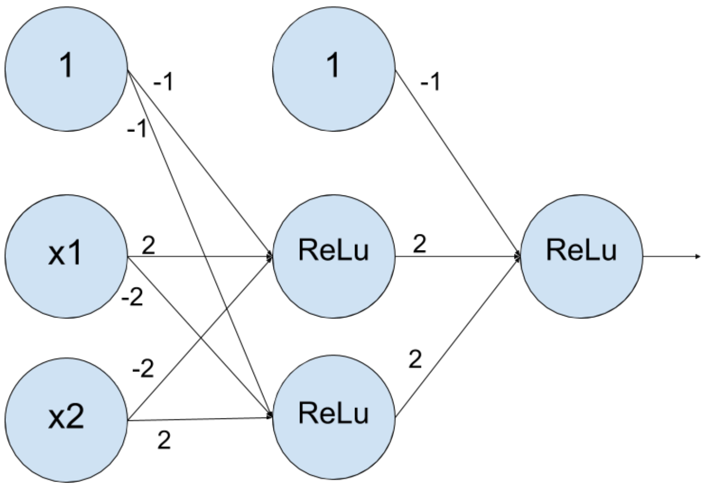
$$\text{XOR}(1,0) \geq 1$$
$$\text{XOR}(0,1) \geq 1$$
$$\text{XOR}(0,0) = \text{XOR}(1,1) = 0$$

For the purposes of this problem, we say that a network f computes the XOR function if $f(x_1, x_2) = \text{XOR}(x_1, x_2)$ when $x_1 \in \{0,1\}$ and $x_2 \in \{0,1\}$ (as described in the examples above).

Explain why a network with fewer layers than the number you specified cannot compute XOR.

Solution C.ii: *Note that the output classes $\{0,1\}$ of XOR are not linearly separable with respect to x_1, x_2; in (x_1, x_2) coordinates, the points $(0,0),(1,1)$ (corresponding to an output of 0) and $(1,0),(1,1)$ (corresponding to an output of 1) are not linearly separable. Thus, we can't compute XOR using a single fully-connected ReLU layer.*

However, we can implement XOR using two fully connected layers, as shown below:



The first layer consists of two units; one that computes x_1 AND (NOTx_2) and one that computes x_2 AND (NOTx_1). The second (output) layer computes the OR of the outputs of the first layer.

2 Depth vs Width on the MNIST Dataset [25 Points]

Relevant Materials: Lectures on Deep Learning

MNIST is a classic dataset in computer vision. It consists of images of handwritten digits (0 - 9) and the correct digit classification. In this problem you will implement a deep network using Keras to classify MNIST digits. Specifically, you will explore what it really means for a network to be ”deep”, and how depth vs. width impacts the classification accuracy of a model. You will be allowed at most N hidden units, and will be expected to design and implement a deep network that meets some performance baseline on the MNIST dataset.

Problem A: Installation [2 Points]

Before any modeling can begin, Tensorflow and Keras must be installed. Tensorflow is a mathematical optimization framework that is widely used in machine learning. Keras is a python package that provides an easy programming interface for constructing neural networks, and uses Tensorflow under the hood.

To install Tensorflow, follow the steps on `https://www.tensorflow.org/get_started/os_setup`. We recommend using the Anaconda install instructions if you installed python via the Anaconda distribution, or using the Pip install instructions if you didn’t.

To install Keras, we recommend you simply use **pip install keras**. If you want to install Keras under the version of Python you get by running **python3**, you can run **python3 -m pip install keras**. For further installation instructions look at `https://keras.io/#installation`.

Once you have finished installing, write down the version numbers for both Tensorflow and Keras that you have installed.

Solution A: *Should be some version numbers for both tensorflow and keras. For example:*

Keras: ’1.2.1’

Tensorflow: ’0.12.1’

Problem B: The Data [3 Points]

Load the MNIST dataset using Keras; see the problem 2 sample code for how.

i. [1 points]: Describe the input data. What are the dimensions of the images? What do the values in each array index represent? You can use the **imshow** function in matplotlib if you’d like to see the actual pictures (see the sample code).

Solution B.i: *Initial data: each image is 28x28, with pixel values ranging from 0-255 (larger numbers represent a darker shade). There are 60000 training samples and 10000 testing samples.*

ii. [2 points]: The labels in the data loaded via Keras are values between 0 and 9 corresponding to the digit represented by each image. This is nice and readable for we as humans, but to a neural network it might seem like an image containing a 9 is somehow "bigger" than one containing a 0, and that an image containing an 8 is somehow "closer" to one containing a 9 than an image containing a 1.

To convert our dataset to a form suitable for multiclass classification, we use one-hot encoding. This encoding scheme transforms each label y into a length-10 binary vector v where $v[y] = 1, v[i] = 0$ for $i \neq y$. For example, the the one-hot encoding of label 3 is the vector $[0, 0, 0, 1, 0, 0, 0, 0, 0, 0]$.

Use the keras function **keras.utils.np_utils.to_categorical** to one-hot encode the labels of the entire dataset. Also, reshape your inputs to be a single vector instead of a 2-dimensional image using **numpy.reshape**. (To use these functions properly, read the keras and/or numpy documentation – reading documentation is the most important software engineering skill!)

What is the new shape of your training input?

Solution B.ii:

To categorize labels:

y_train = keras.utils.np_utils.to_categorical(y_train, 10)

To reshape input data:

X_train = X_train.reshape(60000, 784)

X_test = X_test.reshape(10000, 784)

Each training input is now a vector of length 784.

Problem C: Modeling Part 1 [8 Points]

Using Keras's "Sequential" model class, build a deep network to classify the handwritten digits. You may **only** use the following layers:

- **Dense:** A fully-connected layer
- **Activation (ReLU):** Sets negative weights to 0
- **Activation (Softmax):** Sets highest weight to 1, rest to 0
- **Dropout:** Takes some probability and at every iteration sets weights to zero at random with that probability (effectively regularization)

A sample network with 20 hidden units is in the sample code file. (Note: Activation, Dropout, and your last Dense(NB_CLASSES) layer do not count toward your hidden unit count, because the final layer is "observed" and not *hidden*.)

Use categorical cross entropy as your loss function. There are also a number of optimizers you can use (an optimizer is just a fancier version of SGD), and feel free to play around with them, but RMSprop and Adam are the most popular and will probably work best. You also should find the batch size and number of epochs that give you the best results (default is batch size = 32, epochs=10).

Look at the sample code to see how to compile and train your model. Keras should make it very easy to tinker with your network architecture.

Your task. Using at most 100 hidden units, build a network using only the allowed layers that achieves test accuracy of at least 0.975. Turn in the code of your model as well as the best test accuracy that it achieved.

Hint: for best results on this problem and the two following problems, normalize the input vectors by dividing the values by 255 (as the pixel values range from 0 to 255).

Solution C:

```
model = Sequential()
model.add(Dense(100, input_shape=(784,)))
model.add(Activation('relu'))
model.add(Dense(10))
model.add(Activation('softmax'))
Test accuracy: 0.9777
```

Problem D: Modeling Part 2 [6 Points]

Repeat problem C, except that now you may use 200 hidden units and must build a model with at least 2 hidden layers that achieves test accuracy of at least 0.98.

Solution D:

```
model = Sequential()
model.add(Dense(150, input_shape=(784,)))
model.add(Activation('relu'))
model.add(Dropout(0.2))
model.add(Dense(50))
model.add(Activation('relu'))
model.add(Dropout(0.2))
model.add(Dense(10))
model.add(Activation('softmax'))
Test accuracy: 0.981
```

Problem E: Modeling Part 3 [6 Points]

Repeat problem C, except that now you may use 1000 hidden units and must build a model with at least 3 hidden layers that achieves test accuracy of at least 0.983.

Solution E:

```
model = Sequential()
model.add(Dense(700, input_shape=(784,)))
model.add(Activation('relu'))
model.add(Dropout(0.3))
model.add(Dense(200))
model.add(Activation('relu'))
model.add(Dropout(0.2))
model.add(Dense(100))
model.add(Activation('relu'))
model.add(Dropout(0.1))
model.add(Dense(10))
model.add(Activation('softmax'))
Test accuracy: 0.9847
```

3 Convolutional Neural Networks [40 Points]

Relevant Materials: Lecture on CNNs

Problem A: Zero Padding **[5 Points]**

Consider a convolutional network in which we perform a convolution over each 8×8 patch of a 20×20 input image. It is common to zero-pad input images to allow for convolutions past the edges of the images. An example of zero-padding is shown below:

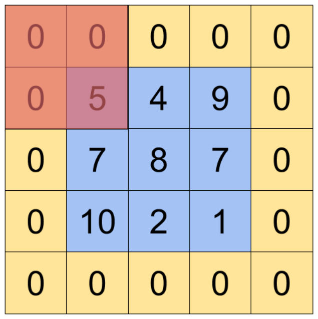


Figure: A convolution being applied to a 2×2 patch (the red square) of a 3×3 image that has been zero-padded to allow convolutions past the edges of the image.

What is one benefit and one drawback to this zero-padding scheme (in contrast to an approach in which we only perform convolutions over patches entirely contained within an image)?

Solution A: *Let's consider the case in which we do not zero-pad our images - since pixels on the border of an image belong to fewer 8×8 image patches than those near the image center, they influence fewer of the convolutional layer's outputs and are therefore underrepresented.*

Zero-padding addresses this issue, resulting in new 8×8 image patches consisting of border pixels and padding pixels. However, since the padding pixels are zero-valued, the border pixels may overinfluence convolutions performed over these new image patches.

Problem B [5 points]: 5 x 5 Convolutions

Consider a single convolutional layer, where your input is a 32×32 pixel, RGB image. In other words, the input is a $32 \times 32 \times 3$ tensor. Your convolution has:

- Size: $5 \times 5 \times 3$
- Filters: 8
- Stride: 1
- No zero-padding

i. [2 points]: What is the number of parameters (weights) in this layer, including a bias term?

Solution B.i:

There are 8 x 5 x 5 x 3 + 8 = 608 weights. This is because for each input in each 5 x 5 x 3 window, we want to create 8 features. Also each of the 8 filters has a bias term.

ii. [3 points]: What is the shape of the output tensor?

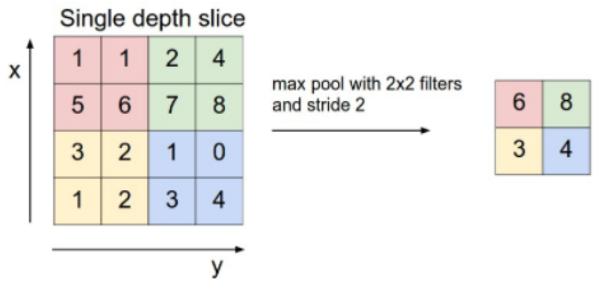
Solution B.ii:

Since there is no zero padding, the sliding window goes over the image (32 - 4) (32 - 4) times. For each filter a vector of 8 features is created. Thus the output has shape 28 x 28 x 8.

Problem C [10 points]: Max/Average Pooling

Pooling is a downsampling technique for reducing the dimensionality of a layer's output. Pooling iterates across patches of an image similarly to a convolution, but pooling and convolutional layers compute their outputs differently: given a pooling layer B with preceding layer A, the output of B is some function (such as the max or average functions) applied to patches of A's output.

Below is an example of max-pooling on a 2-D input space with a 2×2 filter (the max function is applied to 2×2 patches of the input) and a stride of 2 (so that the sampled patches do not overlap):



Average pooling is similar except that you would take the average of each patch as its output instead of the maximum.

Consider the following 4 matrices:

$$\begin{bmatrix}1&1&1&0\\1&1&1&0\\1&1&1&0\\0&0&0&0\end{bmatrix}, \begin{bmatrix}0&1&1&1\\0&1&1&1\\0&1&1&1\\0&0&0&0\end{bmatrix}, \begin{bmatrix}0&0&0&0\\0&1&1&1\\0&1&1&1\\0&1&1&1\end{bmatrix}, \begin{bmatrix}0&0&0&0\\1&1&1&0\\1&1&1&0\\1&1&1&0\end{bmatrix}$$

i. [3 points]:

Apply 2×2 average pooling with a stride of 2 to each of the above images.

Solution C.i:

$$\begin{bmatrix}1&.5\\.5&.25\end{bmatrix}, \begin{bmatrix}.5&1\\.25&.5\end{bmatrix}, \begin{bmatrix}.25&.5\\.5&1\end{bmatrix}, \begin{bmatrix}.5&.25\\1&.5\end{bmatrix}$$

ii. [3 points]:

Apply 2×2 max pooling with a stride of 2 to each of the above images.

Solution C.ii:

$$\begin{bmatrix}1&1\\1&1\end{bmatrix}$$

for all 4 matrices.

iii. [4 points]:

Consider a scenario in which we wish to classify a dataset of images of various animals, taken at various angles/locations and containing small amounts of noise (e.g. some pixels may be missing). Why might pooling be advantageous given these distortions in our dataset?

Solution C.iii:

Pooling would be advantageous because 1) it will remove the noise such as missing pixels and 2) it will provide translational invariance which will allow for the model to work just as well whether the picture of the animal is *in the top left or middle of the image.*

Problem D [20 points]: Keras implementation

Using Keras's "Sequential" model class as you did in 2C, build a deep *convolutional* network to classify the handwritten digits in MNIST. You are now allowed to use the following layers (but **only** the following):

- **Dense:** A fully-connected layer
  - In convolutional networks, Dense layers are typically used to knit together higher-level feature representations.
  - Particularly useful to map the 2D features resulting from the last convolutional layer to categories for classification (like the 1000 categories of ImageNet or the 10 categories of MNIST).
  - Inefficient use of parameters and often overkill: for A input activations and B output activations, number of parameters needed scales as $O(AB)$.
- **Conv2D:** A 2-dimensional convolutional layer
  - The bread and butter of convolutional networks, conv layers impose a translational-invariance prior on a fully-connected network. By sliding filters across the image to form another image, conv layers perform "coarse-graining" of the image.
  - Networking several convolutional layers in succession helps the convolutional network knit together more abstract representations of the input. As you go higher in a convolutional network, activations represent pixels, then edges, colors, and finally objects.
  - More efficient use of parameters. For N filters of $K \times K$ size on an input of size $L \times L$, the number of parameters needed scales as $O(NK^2)$. When N, K are small, this can often beat the $O(L^4)$ scaling of a Dense layer applied to the L^2 pixels in the image.
- **MaxPooling2D:** A 2-dimensional max-pooling layer
  - Another way of performing "coarse-graining" of images, max-pool layers are another way of ignoring finer-grained details by only considering maximum activations over small patches of the input.
  - Drastically reduces the input size. Useful for reducing the number of parameters in your model.
  - Typically used immediately following a series of convolutional-activation layers.
- **BatchNormalization:** Performs batch normalization (Ioffe and Szegedy, 2014). Normalizes the activations of previous layer to standard normal (mean 0, standard deviation 1).
  - Accelerates convergence and improves performance of model, especially when saturating non-linearities (sigmoid) are used.
  - Makes model less sensitive to higher learning rates and initialization, and also acts as a form of regularization.

  - Typically used immediately before nonlinearity (Activation) layers.

- **Dropout:** Takes some probability and at every iteration sets weights to zero at random with that probability
  - An effective form of regularization. During training, randomly selecting activations to shut off forces network to build in redundancies in the feature representation, so it does not rely on any single activation to perform classification.
- **Activation (ReLU):** Sets negative weights to 0
- **Activation (Softmax):** Sets highest weight to 1, rest to 0
- **Flatten:** Flattens any tensor into a single vector (required in order to pass a 2D tensor output from a convolutional layer as input into Dense layers)

Your tasks. Build a network with only the allowed layers that achieves **test accuracy of at least 0.985**. You are required to use categorical cross entropy as your loss function and to train for 10 epochs with a batch size of 32. Note: your model must have fewer than 1 million parameters, as measured by *'model.count_-params()'*. Everything else can change: optimizer (RMSProp, Adam, ???), initial learning rates, dropout probabilities, layerwise regularizer strengths, etc. You are not required to use all of the layers, but *you must have at least one dropout layer and one batch normalization layer in your final model*. Try to figure out the best possible architecture and hyperparameters given these building blocks!

In order to design your model, you should train your model for 1 epoch (batch size 32) and look at the final **test accuracy** after training. This should take no more than 10 minutes, and should give you an immediate sense for how fast your network converges and how good it is.

Set the probabilities of your dropout layers to 10 equally-spaced values $p \in [0, 1]$, train for 1 epoch, and report the final model accuracies for each.

You can perform all of your hyperparameter validation in this way: vary your parameters and train for an epoch. After you're satisfied with the model design, you should train your model for the full 10 epochs.

In your submission. Turn in the code of your model, the test accuracy for the 10 dropout probabilities $p \in [0, 1]$, and the final test accuracy when your model is trained for 10 epochs. We should have everything needed to reproduce your results.

Discuss what you found to be the most effective strategies in designing a convolutional network. Which regularization method was most effective (dropout, layerwise regularization, batch norm)?

Do you foresee any problem with this way of validating our hyperparameters? If so, why?

Hints:

- You are provided with a sample network that achieves a high accuracy. Starting with this network, modify some of the regularization parameters (layerwise regularization strength, dropout probabilities) to see if you can maximize the test accuracy. You can also add layers or modify layers (e.g.

changing the convolutional kernel sizes, number of filters, stride, dilation, etc.) so long as the total number of parameters remains under the cap of 1 million.

- You may want to read up on successful convolutional architectures, and emulate some of their design principles. Please cite any idea you use that is not your own.
- To understand the input and output tensor shapes of your model in order to develop better models, you can access each layer in the list *'model.layers'*, and print *'layer.input_shape'* and *'layer.output_shape'*.
- **Make sure to normalize the input vectors as in 2C (dividing all values by 255).**
- Dense layers take in single vector inputs (ex: *(784,)*) but Conv2D layers take in tensor inputs (ex: *(28, 28, 1)*): width, height, and channels. You will need to reshape the training/test X to a 4-dimensional tensor (ex: *(num_examples, width, height, channels)*) using *'np.reshape'*. For the MNIST dataset, *channels=1*. Typical color images have 3 color channels, 1 for each color in RGB.
- If your model is running slowly on your CPU, try making each layer smaller and stacking more layers so you can leverage deeper representations.
- Other useful CNN design principles:
  - CNNs perform well with many stacked convolutional layers, which develop increasingly large-scale representations of the input image.
  - Dropout ensures that the learned representations are robust to some amount of noise.
  - Batch norm is done after a convolutional or dense layer and immediately prior to an activation/nonlinearity layer.
  - Max-pooling is typically done after a series of convolutions, in order to gradually reduce the size of the representation.
  - Finally, the learned representation is passed into a dense layer (or two), and then filtered down to the final softmax layer.

Solution D:

Please see the provided solution Jupyter notebook for this problem. Sample test accuracies for 10 dropout probabilities equally spaced in $[0, 1]$*:*

[0.99309999999999998, 0.99139999999999995, 0.99250000000000005, 0.99060000000000004, 0.98870000000000002, 0.98999999999999999, 0.98950000000000005, 0.98640000000000005, 0.98170000000000002, 0.96189999999999998]

Any justification for design strategies, so long as they seem well-justified, suffice. A strong answer should discuss higher-level feature representations, effects of regularization, etc.

Key point. *As stated in the problem, we validate our hyperparameters by looking at the final test accuracy of the model under consideration. This has the tendency to bleed in information about the test set into the design of the convolutional network. Typically, this problem is dealt with by maintaining a separate* validation *set that can either be held out from the training set, or cross-validated. In this scheme, the test set is* ***only used once we***

have finalized our model design and hyperparameters.