Keaun Moughari

CAP5619

Professor Liu

1 March 2022

1. Task 1

The task at hand is to design three different neural networks, all of which are to consist of at least four layers with each model making use of both a rectified and sigmoidal/tanh activation function (each used at least once). All models are trained using stochastic gradient descent (SGD) with a batch size of 32 for 10 epochs. The last layer of every model is a fully-connected layer, the output of which is used as the input to a softmax classifier. Cross-entropy loss is then used on the prediction and label of the corresponding input to measure the error of the model.

1. Fully Connected Network

The first is a feedforward network composed of fully connected layers (including the output layer), where each neuron in one layer is connected to every neuron in the next layer. The objective of these networks is to classify handwritten digits of the USPS dataset by mapping a 256-dimensional (16x16) input space to a 10-dimensional output space. The nature of the task can make it difficult for a feedforward network to perform optimally due to the character of the data it will be accepting as input. That is, the digits were written by, presumably, thousands of different people with vastly different writing styles, degrees of care, and pencil/pen pressure – as such, each digit of the same class will vary in its size, slant, line quality, and other distinctive features of handwritten numerals. These differences induce a high degree of variability in the training data that the model must be able to accommodate.

This line of logic leads to the most important consideration of designing the architecture: the size of the hidden layers and the depth of the network. Speaking to the former, if the number of connections is too small, then it is unlikely that the model will successfully learn the training set (which is only 7291 digits). But if the number of connections is too large, the model will likely overfit the data and will not be able to generalize to out-of-distribution data without a bigger training set. My intuition, given my current understanding of feedforward networks, is that each layer will encode another feature of the input into its output; starting with low-level features like presence of pixel values in the earlier layers, then moving to higher-level features such as edges, connection of edges, etc. As the representation at each layer becomes more rich with

higher-level features, the number of combinations between features will grow and it will help to have more neurons at each layer to capture that variability. With that being said, I begin with five layers (including the output layer) where each subsequent layer has more neurons than the previous – its shape being: [64 – 128 – 256 – 4096 - 10].

As for assignment of activation functions, I believe that the rectified unit (ReLU) will induce sparsity and illuminate the relevant connections very quickly. Because of the way the input has been normalized (between -1 and 1), the negative values correspond to the background and will go to 0 after the first layer. That being the case, I opt to use ReLU in the first 3 layers and the hyperbolic tangent activation in the fourth layer.

This model achieves accuracy percentages of 94/90 (train/test).

2. Locally Connected Network

The second of the three is a network composed of locally connected layers, where each neuron is connected to a local neighborhood in the next layer and weights are not shared in the first three layers. This resembles the fully connected layer slightly in that each connection has its own parameter, but the receptive field of every neuron in the layer is restricted to a local few of the previous layer. This inductive bias is complementary to the task at hand because two-dimensional images have a strong local structure and local features can be combined early on in order to extrapolate to higher-level abstractions.

To increase speed and minimize number of parameters, I decide on three locally connected layers with 3x3 kernels, each producing 8 feature maps, and two fully connected layers; the first of the two having 2048 neurons, and the second being an output layer (10 neurons) that is fed into a softmax classifier. Following the same logic as before, I use ReLU activations in all hidden layers besides the third locally connected layer, which uses a hyperbolic tangent activation function.

This model achieves accuracy percentages of 16/18 (train/test) even after trying different learning rates, as the professor has suggested, but I figured out doing Task 2 that it is because of a problem with the *keras* implementation itself. I explain more in my note at the end of Task 2, but this is all to say that the problem doesn't have to do with the design of the architecture.

3. Convolutional Network

In a similar fashion, the convolutional network is composed of three convolutional layers (*i.e.*, shared weights) [L1: 3x3 kernel, 8 feature maps, L2: 3x3 kernel, 16 feature maps, L3: 5x5 kernel, 16 feature maps], a fully connected fourth layer (2048 neurons), and a fully connected output layer (10 neurons) with a softmax classifier. And just as the LCN, the CNN will have a hyperbolic tangent activation function after the third convolutional layer and the rest will be ReLU activation functions.

This model achieves percentages of 93/88 (train/test) using the previously stated configurations.

2. Task 2

1. Parameter Initialization

Even before giving much thought to it, the way in which parameters are initialized in a model seems like one of the most important design choices, as the network is being optimized with respect to these parameters and the gradients that inform that optimization are influenced by what values the parameters have. There are a few general setups that seem obvious in comparing: setting the weights to 0, setting them too low, setting them too high, and picking from a uniform distribution.

The first option isn't worth considering as a choice but for sake of analysis, a value of 0 across the board would produce a network in which there would be absolutely no learning when using a ReLU or tanh activation, as the gradient would be 0. Or, every neuron would follow the same error signal and thus learn the same feature if the activation function is non-zero for any 0 input (sigmoidal functions). In fact, any strategy involving initialization of a constant would result in an equal division of influence among the neurons and would cripple the set of expressible functions substantially, destroying the representational power of the model.

As for setting the weights too small, the influence on the output might be too low and provide too small of gradients, making learning too slow or resulting in the vanishing gradient problem, where the optimization function gets stuck in local minima and converges on a non-optimal value. Inversely, setting the weights too high will most likely lead to exploding gradients and result in the model diverging to oscillate between some minimum value.

The last variation of picking from a uniform distribution seems to me to be the best choice of setting the initial value for the weights, as the variability will be give the neurons more of chance to learn different things from the input. **That being the case, it will be the method I use for each of my three networks.** The only factor left in the design is what range the distribution should have to pick from, which will be different for each network.

1. Feedforward Network

   For the range of the distribution for the first network's weights, I've decided to make them between $\left[-\frac{1}{\sqrt{n}}, \frac{1}{\sqrt{n}}\right]$, where n is the number of neurons in the previous layer (and the square-root is just for normalizing the value).

   The implementation observes three cases: learning is effective, learning is too slow, and learning is too fast.

   - Learning is effective – $\left[-\frac{1}{\sqrt{n}}, \frac{1}{\sqrt{n}}\right]$

     This range of initialization values produces a network which achieves accuracy percentages of 95/90 (train/test).

   - Learning is too fast – [2E-1, 8E-1]

     This range of values, which are larger than optimal, cause the model to learn too quickly, diverging to oscillate between vastly different terrain of the loss surface. It seems this is because a disproportional amount of influence is given to a select number of neurons, exploding the gradients and resulting in large updates that transform the loss surface in an unstable manner. The loss of the model fluctuates in the first and second epoch between 3 and 100, before settling into some sub-optimal local minimum around a loss of 2.3. The end result is a network that achieves accuracy percentages of 16/18 (train/test).

   - Learning is too slow – [-3E-2, 3E-2]

     This range of values, which are smaller than optimal, cause the model to learn too slowly and get trapped in some local minimum of the loss surface. It seems this is because the weights aren't substantial enough

and produce gradients that are too small; which, in turn, doesn't provide enough momentum to make it out of the pockets of the loss surface. The loss of this model can't seem to make it out of the 1.5-2.2 range, getting stuck around accuracy percentages of 29/31 (train/test).

2. Locally Connected Network

For the locally connected layers, I've decided to use a range of $\left[-\alpha \frac{1}{\sqrt{n}}, \alpha \frac{1}{\sqrt{n}}\right]$, where

$$n = \frac{\text{input/output shape}^2 \times \text{kernel width} \times \text{kernel height}}{\text{\# output filters}}$$

and

$$\alpha = \sqrt{\text{kernel height-or-width}}$$

The latter being a scaling factor to bring the initial values into an optimal range, which seems to be near the order of 2E-1 – 5E-1. Additionally, the nature of both the locally connected and convolutional networks (inducing local receptive fields by constraining connectivity between neurons) make me think that the networks will be less sensitive to what the range of the weights are compared to the feedforward network, so long as they are able to 'break symmetry' enough to learn different features of the input.

- Learning is effective – $\left[-\alpha \frac{1}{\sqrt{n}}, \alpha \frac{1}{\sqrt{n}}\right]$

  This range of initialization values produces a network which achieves accuracy percentages of 98/94 (train/test). Its accuracy on both the training and testing sets are higher than feedforward network above, but its generalization performance is slightly worse than the feedforward network.

- Learning is too fast – [-2, 2]

This range of values seem to cause the model to learn too quickly, although it is difficult to make a distinction between effective and too-fast at these percentages since variations become more subtle. This range of values results in a network with accuracy percentages of 100/93 (train/test), where the generalization performance suffers as a result of presence of larger weights.

- Learning is too slow – [-3E-2, 3E-2]

  Values in this range result in a network with percentages of 87/82 (train/test). As seen previously, the small weights produce small gradients and hinder the model from escaping some local minimum on the loss surface.

I should note that the LCN using *keras*'s implementation does not perform much better than random chance (16/18) using its default initialization scheme (glorot uniform initialization). I'm not entirely sure why this may be other than the fact that *keras*'s implementation of the locally connected layer requires passing an initializer function explicitly, though they don't say so in their documentation. I know other people have had this same problem and this might be the reason.

3. Convolutional Network

Similar to the locally connected network and for purpose of comparison, the convolutional network will share the same initialization strategy as stated above.

- Learning is effective – $\left[-\alpha \frac{1}{\sqrt{n}}, \alpha \frac{1}{\sqrt{n}}\right]$

  This range of initialization values produces a network which achieves accuracy percentages of 96/92 (train/test). In addition to the accuracy metrics of both this network and the locally connected variant, the generalization performance is slightly better than the feedforward network.

- Learning is too fast – [-2, 2]

This range of values seem to cause the model to learn too quickly, and shallowly, as it achieves percentages of 96/90 (train/test) – the same as in the LCN.

- Learning is too slow – [-4E-2, 7E-2]

  This range of values produce gradients that are too small for the network to make it out of sub-optimal local minimum on the loss surface, although its generalization performance is better than both of the other strategies attempted; it achieves accuracy percentages of 88/85 (train/test).

2. Learning Rate

The learning rate will be multiplied by the gradient of the loss function. That being the case, it should be clear what the relationship between choice of learning rate and learning is. A learning rate that is too small will result in small updates and might tether the model to some sub-optimal local minimum of the loss surface. Inversely, a learning rate that is too large will cause large updates, overshooting into divergence and fluctuating between some local minimum.

The initial learning rate chosen for all three models when completing the first task was 5E-2, and it seemed to be adequate, but I think there is still some room to move before getting into 'too large' territory. Also, I feel as though the smaller the set of functions the network is able to express, the less complex the loss surface will be, or at least it might be more smoothed than an unrestricted connection scheme (feedforward networks). And as a result, I'll be able to move to higher learning rates in the locally connected and convolutional network than I will the fully connected network; so, my optimization will begin with a learning rate of 1E-1 for the feedforward network and 4E-1 for the other two.

1. Feedforward Network

- Learning is effective – 1E-1

  The first choice of 1E-1 was an effective rate and seemingly optimal choice, as it outperformed the initial learning rate of 5E-2 in both accuracy metrics and generalization performance. It achieved percentages

of 96/92 (train/test). I also tried 2E-1, which got a higher training accuracy, but resulted in lower generalization performance.

- Learning is too fast – 5E-1

  This learning rate results in the network bouncing around loss values of 3-900 in the first 3 epochs before settling on a 2.9 loss and accuracy percentages of 19/17 (train/test). Its divergence is anything but subtle and behaves just as a high learning rate should.

- Learning is too slow – 1E-2

  Just as the above behavior meets expectations, this learning rate does the same. It settles into percentages of 79/75 (train/test), becoming stuck in some local minimum of the loss surface as expected.

2. Locally Connected Network

   - Learning is effective – 5E-1

     Surprisingly, the initial choice of 4E-1 still left a little room for going higher; it produced percentages of 100/95 (train/test). The rate of 5E-1 (which was unstable in the case of the feedforward network) performed even better with 100/96 (train/test), improving on generalization as well.

   - Learning is too fast – 8E-1

     Divergence was immediately apparent, producing 69/18 (train/test) in the 1st epoch and then settling into 16/18 (train/test) on the 2nd epoch.

   - Learning is too slow – 1E-2

     The same rate for slow learning in the feedforward network also tethered this network to sub-optimal local minima in the loss landscape (comparatively). It achieved 94/90 (train/test) – it seems the floor and ceiling of learning rates is farther apart for the locally connected network than the feedforward network, as initially thought.

3. Convolutional Network

- Learning is effective – 25E-2

  The first choice of 4E-1 was too high for the convolutional network, which reinforces my hypothesis stated earlier, as the set of functions expressible by a LCN are a subset of those expressible by a CNN. The learning rate of 25E-2 achieved percentages of 99/95 (train/test), which are impressively higher than with the initial rate of 5E-2 and also preserve the generalization performance.

- Learning is too fast – 3E-1

  The convolutional network did not seem to be as sporadic and obvious with its divergence as the previous two were. The oscillations were rather constrained to loss values of 1.0-2.4 and the model settled into percentages of 28/29 (train/test).

- Learning is too slow – 1E-2

  Following the previous two networks for slow learning, the rate of 1E-2 was sufficient to strangle performance. The model achieved percentages of 89/85.

3. Batch Size

   In each layer of a network, there is a corresponding distribution for each input that is modified in the training procedure by the noise of the initialization scheme and input of the model. As a result, each layer must constantly readjust to new distribution, with each adjustment being more severe than the previous; said differently, small changes in the earlier layers result in large changes in later layers. Batch normalization is a method that attempts to mitigate this and induce stability in a model by normalizing the input through re-centering and re-scaling.

   The batch size determines how many samples there are per iteration, and in turn, how many updates there are per iteration. The more samples you have, the closer you are to an accurate ‘average’ of the distribution the model is trying to fit.

With that in mind, it stands to reason that larger batch sizes will be complementary to batch normalization, while smaller batch sizes that aren't as representative will stunt the effect of batch normalization and performance will suffer. **In my implementation, a batch size of 8 completely destroys my performance and achieves percentages of 54/50 (train/test), while a batch size of 32 seems to be optimal and achieves percentages of 99/95 (train/test).**

4. Momentum

Bath Size: *32*, LR: *25E-2*

M : *0.00* – 99/95 (train/test)

M: *0.50* – 99/94 (train/test)

M: *0.90* – 16/18 (train/test)

M: *0.99* – 16/18 (train/test)

Above shows the results of different values for momentum using the convolutional network with optimal batch size and learning rate. A momentum of 5E-1 seems to hurt generalization performance, while the other two nonzero values cause the model to diverge in similar manners. I suspect this is because of the nature of momentum on an already optimized network. That is, the model is already navigating the loss landscape perfectly without needing a 'push', and to add some push results in the model to overshoot into sub-optimal regions of the loss surface.

3. Task 3

1. Bagging

The ensemble I used in this task is comprised of 4 LCN's, each trained with a slightly different learning rate, but the same configurations as mentioned above (batch size: 32, my custom initialization scheme, no momentum, etc.). In order to split the training set for each model, I sampled the examples randomly for each model (the *numpy* module I used to generate the random indices for the mask seems to produce a training with at least 4,500 unique samples of the 7,291 samples). To save time in tinkering with the final ensemble, I pretrained the 4 models and saved their final state

to be loaded. The models' individual test accuracies were the following: 94.47% (LR: 55E-2), 94.47% (LR: 56E-2), 94.76% (LR: 5E-1), and 94.92% (LR: 45E-2).

The ensemble uses a majority voting scheme where a prediction is computed for every test example for each model, then the most common answer of the 4 models is used for the prediction of the final model. The final model achieved a test accuracy of 95.07%, increasing the generalization performance only slightly. My intuition is that this might be improved if the sampling mask produced a training set with a lower number of duplicates.

2. Dropout

Dropout is a regularization technique that is motivated by the problem of overfitting in neural networks. The central idea is that if some neurons are randomly 'dropped out', then this unreliability will force neurons in following layers to organize itself with this possibility already encoded and learn what's needed in a more efficient and robust way. This technique inherently lowers the effective capacity of models, thus it may take increasing the number of neurons in order to achieve the same training accuracy. Although, I decide not adjust the number of neurons to accommodate for this for sake of comparison.

The feedforward network is trained with the same configurations as above (learning rate: 1E-1, batch size: 32, custom initialization scheme, no momentum, etc.) and has baseline accuracies of 96/92 (train/test) without use of dropout. I've decided to use dropout after each hidden layer in the model.

- Effective – Dropout of 0.4

  This dropout produces a network with accuracy percentages of 90/90 (train/test). As said previously, the nature of the regularization lowers the effective capacity of the model, and thus, the training accuracy drops by a good deal but the generalization performance is perfect; that is, the model performs just as well on out-of-distribution data as it does on the training set.

- Not effective – Dropout of 0.8

  This dropout value cripples the network, producing percentages of 21/28 (train/test). Because each neuron has an 80%

chance of being pruned at each layer, the model doesn't even have the capacity to fit the training data and results in the accuracy just mentioned.