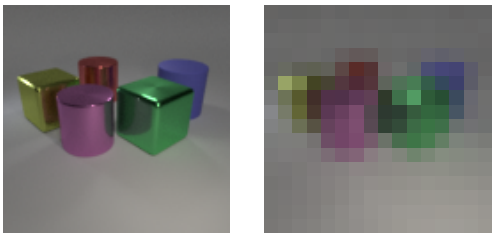
2. **(38 points) Neural Networks:**



(a) Original Image (b) Re-scaled Image

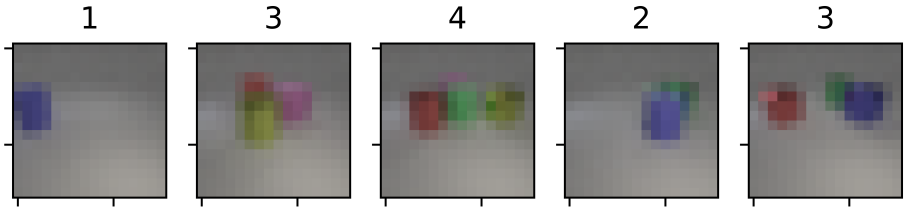


Figure 1: Sample data. The numeric label represents the true number of objects in an image.

In this problem, you will work with the modified version of CLEVR dataset. This is an image dataset consisting of RGBA pixel values of 16×16 size images of objects placed on grey surface. The task is to predict number of objects in an image. The number of objects in an image vary between 1 to 5. Therefore this is a multiclass dataset containing 5 classes. Fig 2 shows some sample examples. The Dataset contains a train and a test set consisting of 10000 and 1000 examples, respectively. Note that the dataset is balanced; i.e. the number of examples is approximately same for all classes in train and test set. Instead of using the standard squared error loss over the logistic function, we will instead use the cross-entropy loss defined over the softmax activation function. Softmax is a generalization of logistic function for the case of multi-valued variables. Given a set of linear inputs $\{net_k^{(L)}\}_{k=1}^r$, applying softmax over it, results in a probability vector of size r, given as $o_1, o_2 \cdots o_r$, such that $\sum_k o_k = 1$, and where $o_k = \frac{e^{net_k^{(L)}}}{\sum_{k'} e^{net_{k'}^{(L)}}}$ Given two distributions P and Q, defined over a random variable y, the cross-entropy between them is defined via the expression: $\sum_k P(y=k) log Q(y=k)$. Then, in our case, the loss $J(\theta)$, is defined as the cross entropy between the true label distribution given as $\tilde{p}(y=k|x) = \mathbb{1}\{k=\tilde{k}\}$, where $\tilde{k}$, is the true label, and the predicted distribution $\hat{p}(y=k|x) = o_k$ (recall that o_k is simply the probability of outputting label k by the network). Then, the cross entropy loss is defined as:

$$J(\theta) = \sum_{k=1}^{r} -\mathbb{1}\{k=\tilde{k}\} log(o_k) \tag{1}$$

The partial derivative of $J(\theta)$, as defined above is given as:

$$\frac{\partial J(\theta)}{\partial net_k^{(L)}} = \begin{cases} (o_k - 1), & \text{if } k = \tilde{k} \\ o_k & \text{otherwise} \end{cases} \tag{2}$$

(a) **(12 points)** Write a program to implement a generic neural network architecture to learn a model for multi-class classification which outputs probability distribution by using softmax. You will implement the backpropagation algorithm (from first principles) to train your network. You should use mini-batch Stochastic Gradient Descent (mini-batch SGD) algorithm to train your network. Use the Cross Entropy Loss over each mini-batch as your loss function (mentioned above in 1). You will use the sigmoid as activation function for the units in the hidden layer (we will experiment with other activation units in one of the parts below) and softmax at output layer to get final predicted probability distribution. Your implementation(including back-propagation) MUST be from first principles and not using any

pre-existing library in Python for the same. It should be generic enough to create an architecture based on the following input parameters:

- Mini-Batch Size (M)
- Number of features/attributes (n)
- Hidden layer architecture: List of numbers denoting the number of perceptrons in the corresponding hidden layer. Eg. a list [100 50] specifies two hidden layers; first one with 100 units and second one with 50 units.
- Number of target classes (r)

Assume a fully connected architecture i.e., each unit in a hidden layer is connected to every unit in the next layer.

(b) **(5 points)** Use the above implementation to experiment with a neural network having a **single** hidden layer. Vary the number of hidden layer units from the set {1, 5, 10, 50, 100}. Set the learning rate to 0.01. Use a mini-batch size of 32 examples. This will remain constant for the remaining experiments in the parts below. To be specific you will have following arguments in the generic neural network:

- $M = 32$
- $n = 1024 \;\; (16 \times 16 \times 4)$
- Hidden layer sizes to be choosen from options
- $r = 5$

Choose a suitable stopping criterion and report it. Read about precision, recall and F (also known as F1) score here. Report the precision, recall and F1 score for each class at different hidden layer units on test data and train data. You could compute these metrics using scikit-learn utility. Plot the average F1 score vs the number of hidden units. Comment your findings.

(c) **(5 points)** In this part we will experiment with the depth of neural network. Vary the hidden layer sizes as $\{\{512\}, \{512, 256\}, \{512, 256, 128\}, \{512, 256, 128, 64\}\}$. Keep learning rate and batch size same as part b. Use same stopping criteria as before and report the precision, recall and F1 score for all variations on test data and train data. Plot the average F1 score vs the network depth.

(d) **(5 points)** Use an adaptive learning rate inversely proportional to number of epochs i.e. $\eta_e = \frac{\eta_0}{\sqrt{e}}$ where $\eta_0 = 0.01$ is the seed value and e is the current epoch number[2] and repeat part c. See if you need to change your stopping criteria. Report your stopping criterion. As before, report the precision, recall and F1 score for each class at different hidden layer depth on test data and train data. Plot the average F1 score vs depth of hidden units. How do your results compare with those obtained in the part above? Does the adaptive learning rate make training any faster? Comment on your observations.

(e) **(6 points)** Several activation units other than sigmoid have been proposed in the literature such as tanh, and RelU to introduce non linearity into the network. ReLU is defined using the function: g(z) = max(0, z). In this part, we will replace the sigmoid activation units by the ReLU for all the units in the hidden layers of the network. You can read about relative advantage of using the ReLU over several other activation units on this blog.

Change your code to work with the ReLU activation unit in the hidden layers. The activations in the final layer still stay softmax. Note that there is a small issue with ReLU that it is non-differentiable at $z = 0$. This can be resolved by making the use of sub-gradients - intuitively, sub-gradient allows us to make use of any value between the left and right derivative at the point of non-differentiability to perform gradient descent see this (Wikipedia page for more details). Repeat the part d with ReLU activation. Report your training and test set precision, recall and F1 score. Plot the average F1 socre vs network depth. Also, make a relative comparison of test set accuracies obtained in part d. What do you observe? Which ones performs better?

(f) **(5 points)** Use MLPClassifier from scikit-learn library to implement a neural network with the same architectures as in part e above. Use Stochastic Gradient Descent as the solver. Note that MLPClassifier only allows for Cross Entropy Loss over the final network output. Set the following parameters:

- hidden_layer_sizes: to be vary according to part c.
- activation: relu
- solver: sgd

[2]One epoch corresponds to one complete pass through the data

- alpha: 0
- batch_size: 32
- learning_rate: invscaling

Keep all other parameter as default. You need to decide the stopping criteria accordingly. Now report the same metrics and plots as of part e. Compare the results with part e, and comment on your observations.