ECE 661 - Homework 4

Michael Li (zl310)

November 8, 2023

I have adhered to the Duke Community Standard in completing this assignment.

Contents

List of Figures

1 True/False Questions (20 pts)

Problem 1.1 (2 pts) Generally speaking, the weight pruning does not intervene the weight quantisation, as they are orthogonal techniques in compressing the size of DNN models.

True. We did this in lab 3 of this assignment. Both can be applied without interfering with each other since the effect on the weight values are different: weight pruning is a form of regularisation (and occasionally can save computation resources with applicable hardware) while weight quantisation attempts to save memory usage.

Problem 1.2 (2 pts) In weight pruning techniques, the distribution of the remaining weights does not affect the inference latency.

False. Non-structured sparsity doesn't bring much speedup on traditional GPUs even with specialised sparse-matrix multiplication algorithms according to Lecture 12 slide 25. Structured sparsity constrains the distribution of the nonzero weights, which may result in shorter inference latency.

Problem 1.3 (2 pts) In deep compression pipeline, even if we skip the quantisation step, the pruned model can still be effectively encoded by the following Huffman coding process as pruning greatly reduces the number of weight variables.

False. Without quantisation, it is unlikely that floats will repeat, so Huffman encoding will not help. Huffman encoding only increases storage efficiency when there are repetitions in the values of the weights. This means that the most commonly appearing weights can be represented in fewer bits to save the most space and a look-up table can be used. With quantisation, there exists repetition of values in the weight matrix, so Huffman encoding will reduce number of bits.

Problem 1.4 (2 pts) Directly using SGD to optimize a sparsity-inducing regulariser (i.e. L-1, Deep-Hoyer etc.) with the training loss will lead to exact zero values in the weight elements, there's no need to apply additional pruning step after the optimization process.

False. The weight values will be *guided* towards 0 and won't be exactly zero. According to lecture 13, slide 19, a final pruning step is required to achieve a sparse model; the pruning step's necessity means that the sparsity-inducing regulariser only makes the weights go very close to zero.

Problem 1.5 (2 pts) Using soft thresholding operator will lead to better results comparing to using L-1 regularisation directly as it solves the "bias" problem of L-1.

False. Soft-thresholding is still "biased" since the larger values always shrink by λ. This causes a loss of variance since the larger weights become uniformily smaller according to lecture 14, slide 9.

Problem 1.6 (2 pts) Group Lasso can lead to structured sparsity on DNNs, which is more hardware-friendly. The idea of Group Lasso comes from applying L-2 regularisation to the L-1 norm of all of the groups.

False. It is the other way round. One should apply L-1 regularisation to the L-2 norms of all the groups according to lecture 13, slide 21. This makes sure that the L-2 norms all shrink to zero due to L-1 regularisation. In any case, applying L-2 regularisation to an expression is worse at making the expression go to zero compared to applying L-1.

Problem 1.7 (2 pts) Proximal gradient descent introduces an additional proximity term to minimize regularisation loss in the proximity of weight parameters. The proximity term allows smoother convergence of the overall objective.

True. According to lecture 14, slide 6, convergence will become smoother. This behaviour is observed in Figure 3 (L-1) and 4 (proximal gradient update). Compare the smoother convergence in Figure 4 with the more squiggly convergence in Figure 3 (vanilla L-1).

Problem 1.8 (2 pts) Models equipped with early exits allows some inputs to be computed with only part of the model, thus overcoming the issue of over-fitting and overthinking.

True. According to lecture 14, slides 27-28, early exits reduces over-fitting and overthinking and improves testing accuracy. Using less layers to identify and predict inputs that are less complicated reduces the chance that the model will mis-predict in the later stages.

Problem 1.9 (2 pts) When implementing quantisation-aware training with STE, gradients are quantised during back-propagation, ensuring that updates are consistent with the quantised weights.

False because the gradients are NOT quantised during back-propagation (lecture 15, slide 10). The full-precision gradients and weight values are always kept when training the model. The quantised weights are used in the forward-propagating step with prediction and loss calculation. The model would become unstable if the gradient values are also quantised.

Problem 1.10 (2 pts) Comparing to quantising all the layers in a DNN to the same precision, mixed-precision quantisation scheme can reach higher accuracy with a similar-sized model.

True, according to lecture 15 slide 28. For example, more "important" layers (those that affect the loss a lot when the their weights are changed, quantitatively determined using the Hessian) require more bits to quantise (lecture 15 slide 35) and fewer quantisation bits are required for layers with more weight values.

2 Lab 1: Sparse optimization of linear models (30 pts)

(a) (4 pts) Theoretical analysis: with learning rate μ, and supposing the weight you have after step k is W^k, derive the symbolic formulation of weight W^{k+1} after step $k+1$ of full-batch gradient descent with $X_i, y_i, i \in \{1, 2, 3\}$.

According to slide 9 lecture 2, the relationship between W^{k+1} and W^k is $W^{k+1} = W^k - \mu \frac{\partial L}{\partial W}\big|_{W=W^k}$.

$$\begin{aligned} W^{k+1} &= W^k - \mu \left.\frac{\partial L}{\partial W}\right|_{W=W^k} = W^k - \mu \left.\frac{\partial}{\partial W}\left[\sum_i \left(X_i W - y_i\right)^2\right]\right|_{W=W^k} \\ &= W^k - \mu \left[\sum_i \left.\frac{\partial}{\partial W}\left(X_i W - y_i\right)^2\right|_{W=W^k}\right] \end{aligned} \tag{1}$$

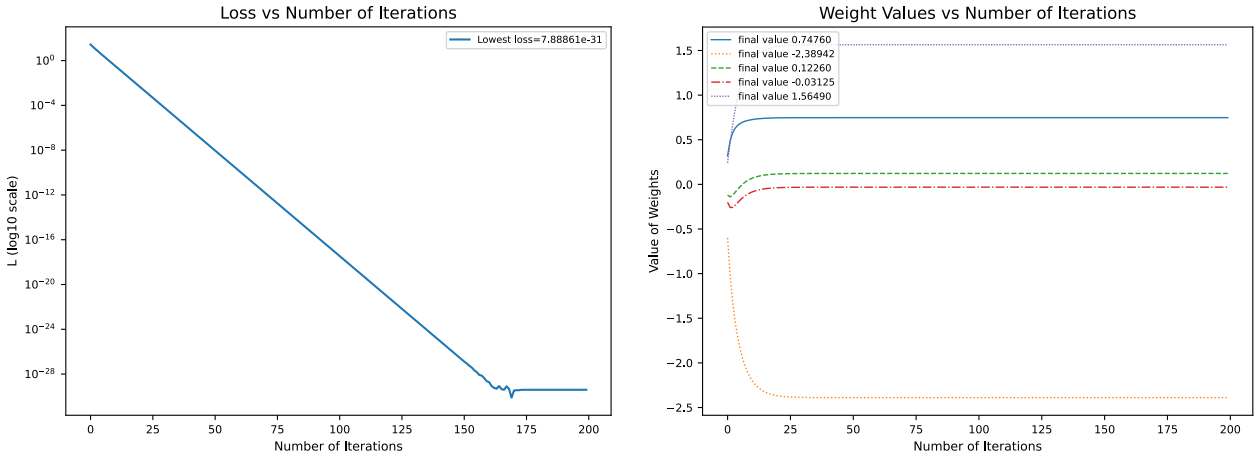
Let $u = X_i W - y_i$ and $g(u) = u^2 = (X_i W - y_i)^2$ where $g : \mathbb{R}^1 \to \mathbb{R}^1$. Using rules of matrix calculus and the numerator layout as explained here (https://en.wikipedia.org/wiki/Matrix_calculus#Numerator-layout_notation),

$$\frac{\partial L}{\partial W} = \sum_i \frac{\partial}{\partial W}\left(X_i W - y_i\right)^2 = \sum_i \frac{\partial g(u)}{\partial u}\frac{\partial u}{\partial W} = \sum_i \left[\left(2 \cdot \left(X_i W - y_i\right) X_i^T\right]\right. \tag{2}$$

Combining Equations 1 and 2,

$$W^{k+1} = W^k - \mu \left[\sum_i \left.\frac{\partial}{\partial W}\left(X_i W - y_i\right)^2\right|_{W=W^k}\right] = \boxed{W^k - \mu \cdot \sum_i \left[2 \cdot \left(X_i W^k - y_i\right) X_i^T\right]} \tag{3}$$

(b) (3 pts) In Python, directly minimize the objective L without any sparsity-inducing regularisation/constraint. Plot the value of log(L) vs. number of steps throughout the training, and use another figure to plot how the value of each element in W is changing throughout the training.



(a) Loss vs Iterations.

(b) Weight Values vs Iterations.

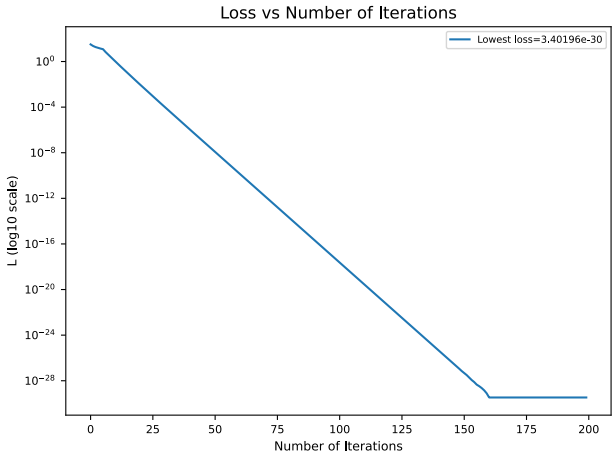
Figure 1: Lab1 part b, plain gradient descent.

```
def no_regularisation_update(in_x, in_y, w_prev, l_1_reg_param=None, lr=None):
    n_r, n_c = in_x.shape
    grad = 0
    for inner in range(n_r):
        grad += 2 * (np.dot(in_x[inner, :], w_prev) - in_y[inner]) * in_x[inner].reshape(-1, 1)
    if lr is not None:
        return w_prev - lr * grad
    else:
        return grad
```

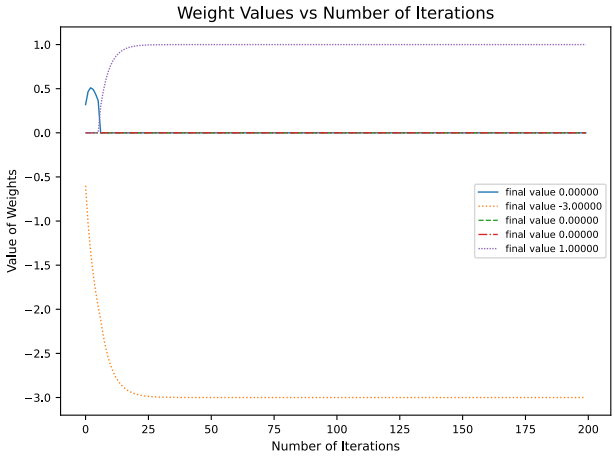
The code snippet above shows my function for calculating gradient descent with no pruning, no regularisation, and no priximal gradient update. The sum in Equation 3 is computed (i.e., the gradient is computed) and the weight is updated using lr. The gradient update function are passed in as the argument `grad_fn` in `run_grad_descent(...)`. Please refer to the PDF of the Jupyter Notebook for more details.

From your result, is W converging to an optimal solution? Is W converging to a sparse solution?
W is converging to an optimal solution since the loss is extremely close to zero at iteration 199 (Figure 1a). However, W isn't converging to a sparse solution since all of the weight values are nonzero (1b).

(c) (6 pts) Since we have the knowledge that the ground-truth weight should have $||W||_0 \leq 2$, we can apply projected gradient descent to enforce this sparse constraint. Redo the optimization process in (b), this time prune the elements in W after every gradient descent step to ensure $||W^l||_0 \leq 2$. Plot the value of log(L) throughout the training, and use another figure to plot the value of each element in W in each step.



(a) Loss vs Iterations.



(b) Weight Values vs Iterations.

Figure 2: Lab1 part c, iterative pruning.

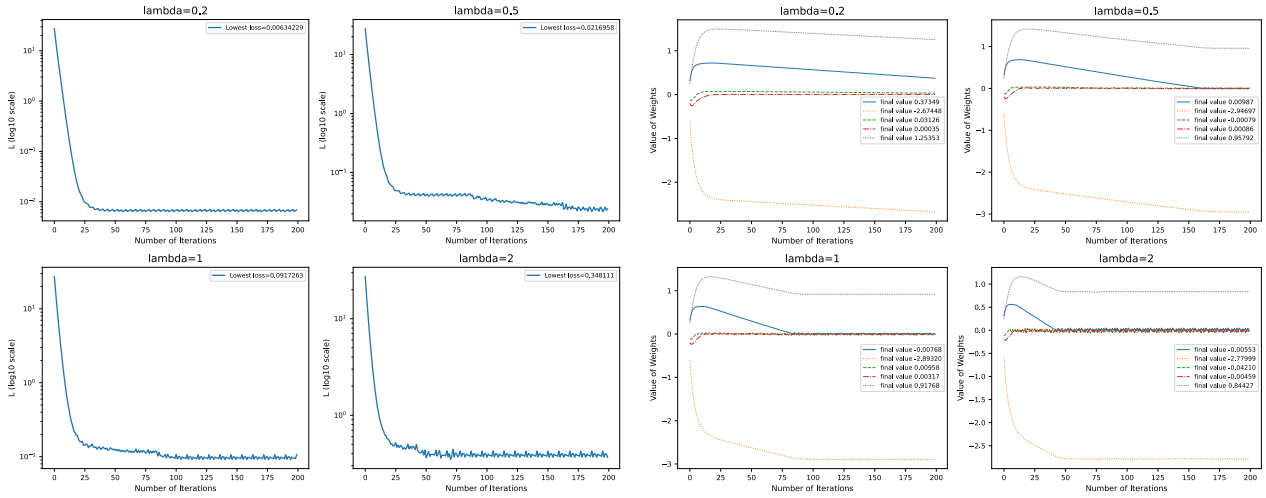
```
def run_grad_descent(w_init, lr, in_x, in_y, epochs, w0_prune=False, thresh=None, l_1_reg_param=None,
                     grad_fn=no_regularisation_update):
    # not shown ...
    for i in range(epochs):
        if not w0_prune and thresh is not None:  # only for lab 1 f, l_1-trimmed
            w_curr = grad_fn(in_x, in_y, w_prev, l_1_reg_param, lr=lr, thresh=thresh)
        else:  # lab 1b, c, d, e
            w_curr = grad_fn(in_x, in_y, w_prev, l_1_reg_param, lr=lr)

        if w0_prune and thresh is not None:  # only for lab 1 c, pruning
            w_curr_sorted_idx = np.argsort(np.abs(w_curr.flatten()))  # sort ascending
            for inner in range(len(w_init) - thresh):
                # len(w_init) - thresh = 5 - 2 = 3, so three smallest weights (absolute value) are
                # forced to zero
                w_curr[w_curr_sorted_idx[inner], 0] = 0  # zero-ing out smallest elements
    # not shown ...
    return # not shown ...
```

The above code snippet highlights the weight pruning process. The for loop is for gradient update. In lab1 part c `grad_fn == no_regularisation_update` (i.e., using vanilla gradient update) and the condition `if w0_prune and thresh is not None` is satisfied. The coordinates in the weight vector with the smallest absolute values are set to zero. Please refer to the PDF of the Jupyter Notebook for more details.

From your result, is W converging to an optimal solution? Is W converging to a sparse solution?

W is converging to an optimal solution (Figure 2a) and W is also converging to a sparse solution since Figure 2b shows that there are only two coordinates in the weight vector are nonzero.

(d) (5 pts) In this problem we apply l_1-regularisation to induce the sparse solution. The minimisation objective therefore changes to $L + \lambda||W||_1$. Please use full-batch gradient descent to minimize this objective, with $\lambda = \{0.2, 0.5, 1.0, 2.0\}$ respectively. For each case, plot the value of log(L) throughout the training, and use another figure to plot the value of each element in W in each step.



(a) Loss vs Iterations.

(b) Weight Values vs Iterations.

Figure 3: Lab1 part d, l_1-regularisation for $\lambda = \{0.2, 0.5, 1.0, 2.0\}$.

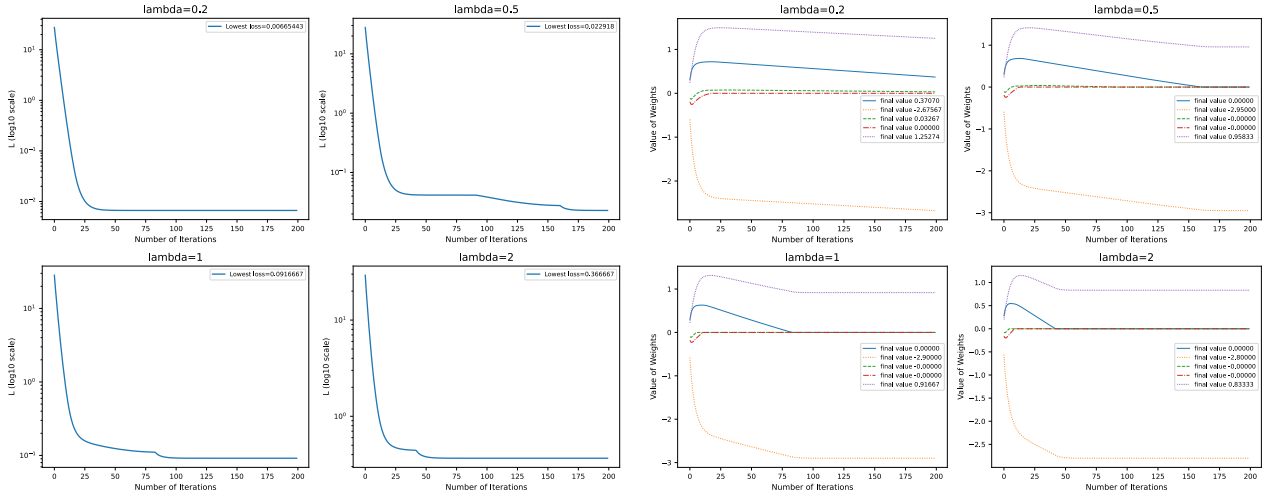
```
def l_1_norm_update(in_x, in_y, w_prev, l_1_reg_param, lr):
    """
    Args: # not shown ...
    Returns: updated weight
    """
    grad = no_regularisation_update(in_x, in_y, w_prev, l_1_reg_param, lr=None) + \
           l_1_reg_param * np.sign(w_prev)
    return w_prev - lr * grad
```

The above code snippet shows the weight-update function used for lab 1 part d. The l_1-norm gradient update adds the gradient of the l_1-norm ($w_{i+1} = \lambda \cdot \text{sgn}(w_i)$) to the gradient of the loss function. Please refer to the PDF of the Jupyter Notebook for more details.

From your result, comment on the convergence performance under different λ.
See Figure 3 for the required plots. The performance of the optimisation process using l_1-norm becomes worse as λ becomes larger. There is significant of oscillation in the loss value as the regularisation parameter increases in magnitude. The loss does not converge to a low value, meaning that the optimisation process is stuck on a local minimum. There are small, sudden decreases in the loss value when $\lambda = \{0.5, 1, 2\}$ in Figure 3a (perhaps because the optimisation process jumped from one local minimum into another local minimum).

The values of the coordinates in the weight vector (part (d)) for $\lambda = 0.5$ are very close to those in the weight vector in part (c) (iterative pruning). However, due to oscillations in the loss, the weight values using l_1-regularisation never end up being equal to the weight values when using iterative pruning, causing the loss to never decrease beyond 10^{-2}. The coordinates in the weight values go to zero more quickly but oscillate more for larger values of λ.

(e) (6 pts) Here we optimise the same objective as in (d), this time using **proximal gradient update**. Recall that the proximal operator of the l_1 regulariser is the soft thresholding function. Set the threshold in the soft thresholding function to $\{0.004, 0.01, 0.02, 0.04\}$ respectively. Plot the value of log(L) throughout the training, and use another figure to plot the value of each element in W in each step. (Hint: Optimizing $L + \lambda||W||_1$ using gradient descent with learning rate μ should correspond to proximal gradient update with threshold $\mu\lambda$)



(a) Loss vs Iterations.

(b) Weight Values vs Iterations.

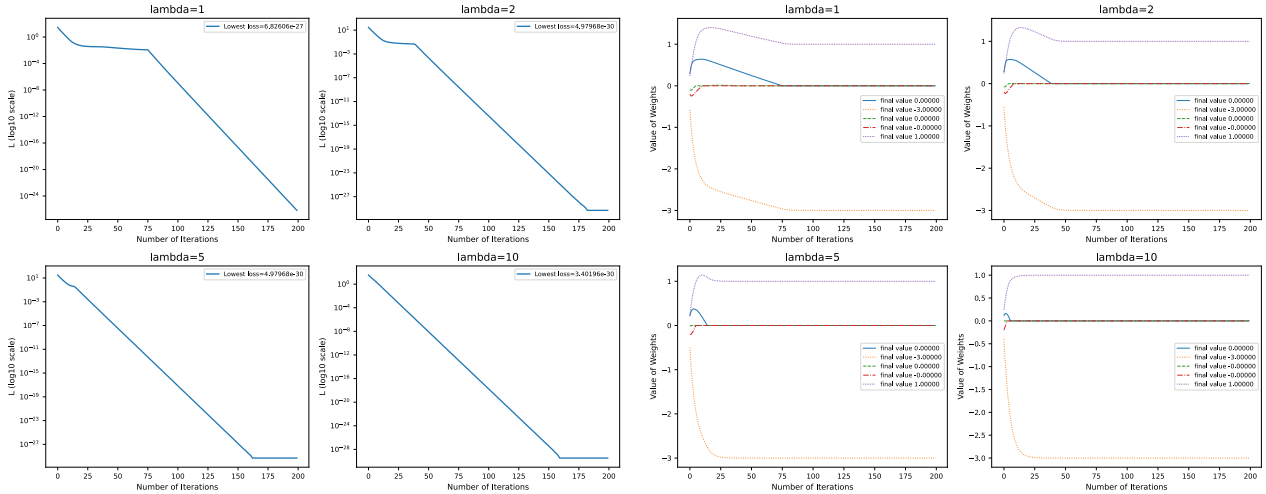
Figure 4: Lab1 part e, proximal gradient update for threshold = $\{0.004, 0.01, 0.02, 0.04\}$.

```
def l_1_norm_proximal_update(in_x, in_y, w_prev, l_1_reg_param, lr, which_weights=None):
    """
    Computes weight using proximal gradient descent (l_1-norm). threshold is l_1_reg_param*lr
    Args: ...
        which_weights: indices of which weights for which proximal update is applied
    Note: threshold = lr * lambda should be in set{0.004, 0.01, 0.02, 0.04}
    Returns: updated weight
    """
    if which_weights is None:
        which_weights = np.array(range(len(w_prev.flatten()))) # which_weights = [0,1,2,3,4]
    weight_tilde = no_regularisation_update(in_x, in_y, w_prev, lr=lr) # main objective
    weight_ret = np.copy(weight_tilde)

    threshold = lr * l_1_reg_param
    for iii in which_weights:  # only updates the weights specified in which_weights
        this_weight = weight_tilde[iii, 0]
        weight_ret[iii, 0] = my_relu(np.abs(this_weight) - threshold) * np.sign(this_weight)
    return weight_ret
```

The code snippet above shows the l_1-norm proximal gradient update. The intermediate weight vector is calculated using vanilla gradient update. Then, each value in the weight vector is edited according to the formula for soft-thresholding. Please refer to the PDF of the Jupyter Notebook for more details.

Compare the convergence performance with the results in (d).
Compared to Figure 3a (l_1), there are less oscillations in Figure 4a (soft-thresholding). However, the shapes and values (ignore the oscillatory behaviour) of the loss curves in Figures 3a and 4a are very similar. Both algorithms fail to find the best and sparsest weight vector in 200 iterations and both manage to exit and then enter local minima.
The values in the weight vector do not oscillate when using proximal gradient update (Figure 4b) compared to part (d) (Figure 3b). Both plots (Figures 4b, 3b) show similar trends in how the weight vector changes over time and neither l_1 nor soft-thresholding manage to produce an optimal sparse solution.

(f) (6 pts) Trimmed l_1 (Tl_1) regularizer is proposed to solve the "bias" problem of l_1. For simplicity you may implement the T l_1 regularizer as applying a l_1 regularisation with strength λ on the 3 elements of W with the smallest absolute value, with no penalty on other elements. Minimize $L + \lambda Tl_1(W)$ using proximal gradient update with $\lambda = \{1.0, 2.0, 5.0, 10.0\}$ (correspond to the soft thresholding threshold values $\{0.02, 0.04, 0.1, 0.2\}$). Plot the value of log(L) throughout the training, and use another figure to plot the value of each element in W in each step. Compare the convergence behaviour of the Trimmed l_1 and the l_1. Also compare the behavior of the early iterations (e.g. first 20) between Tl_1 and iterative pruning.



(a) Loss vs Iterations.

(b) Weight Values vs Iterations.

Figure 5: Lab1 part f, trimmed l_1 for $\lambda = \{1.0, 2.0, 5.0, 10.0\}$.

```
def l1_norm_trimmed_update(in_x, in_y, w_prev, l1_reg_param, lr, thresh):
    """
    thresh: number of elements to keep unchanged when performing proximal l1, returns updated weight
    """
    ascend_idx = np.argsort(np.abs(w_prev.flatten()))
    weight_l1_proximal = l1_norm_proximal_update(in_x, in_y, w_prev, l1_reg_param, lr,
                                                 which_weights=ascend_idx[0:w_prev.shape[0]-thresh])
    return weight_l1_proximal
```

The Tl_1 weight update function is implemented in the code snippet above. Proximal gradient update is applied to the largest values in the weight vector as specified by the parameter `thresh`. Please refer to the PDF of the Jupyter Notebook for more details.

Compare the convergence behaviour of the Trimmed l_1 and the l_1 algorithms (part (d)):
In Figure 5a, we see that the Tl_1 algorithm converges to the optimal weight value, while the l_1 algorithm fails to converge (Figure 3a). There is no oscillation using Tl_1 and the algorithm doesn't get stuck on local minima. The weight values for Tl_1 (Figure 5b) do not oscillate; however, the weight value plots with Tl_1 bear strong resemblance to the corresponding plots with the l_1 algorithm (Figure 3b, look at $\lambda = 1, 2$). This means that Tl_1 does not get stuck and goes to the global minimum without oscillating.

Compare the behavior of the early steps (e.g. first 20) between the Trimmed l_1 and the iterative pruning algorithms (part (c)).
In Figure 2a (pruning), the weight vector spends very little time in a local minimum since the loss value starts decreasing quickly in the first few iterations and heads towards the global minimum. In contrast, the weight vector spends more epochs navigating local minima for lower values of λ in Figure 5a (Tl_1) as evinced by the slow initial decrease of the loss value when $\lambda = 1, 2$ compared to pruning. When $\lambda = 10$ and using Tl_1, (bottom-right of Figure 5a), the weight vector spends less time in a local minimum than with weight pruning since there's no relatively flat part of the loss vs number of iterations plot for $\lambda = 10$ using Tl_1.
For small λ's, the weight values using Tl_1 (top 2 plots in Figure 5b) change more slowly compared to weight pruning. Also, weight values overshoot more with Tl_1 and small λ (purple line of the plots with $\lambda = \{1, 2, 5\}$ in Figure 5b). When λ becomes larger, the weight values with Tl_1 (bottom-right of Figure 5b) approach the sparsest optimal solution more rapidly than with weight pruning (Figure 2b). The purple line in Figure 2b (pruning) starts off at zero, while the purple lines in Figure 5b (Tl_1) do not start zero.

3 Lab 2: Pruning ResNet-20 model (25 pts)

(a) (2 pts) In `hw4.ipynb`, run through the first three code block, report the accuracy of the floating-point pre-trained model.
Test Loss is 0.3231, Test accuracy is $\boxed{0.9150}$.

(b) (6 pts) Complete the implementation of *pruning by percentage function* in the notebook. Here we determines the pruning threshold in each DNN layer by the 'q-th percentile' value in the absolute value of layer's weight element. Use the next block to call your implemented pruning by percentage. Try pruning percentage $q = 0.3, 0.5, 0.7$. Report the test accuracy q.

```
with torch.no_grad():
    # Convert the weight of "layer" to numpy array
    layer_weight = layer.weight.detach().cpu().numpy()
    # Compute the q-th percentile of the abs of the converted array
    percentile = np.percentile(np.abs(layer_weight.flatten()), q)
    # Generate a binary mask same shape as weight to decide which element to prune
    masked_obj = np.ma.masked_greater_equal(x=np.abs(layer_weight), value=percentile, copy=True)
    mask_int = np.ma.getmask(masked_obj).astype(int)
    # Convert mask to torch tensor and put on GPU
    mask_tensor = torch.tensor(mask_int).to(device)
    # Multiply the weight by mask to perform pruning
    assert mask_int.shape == layer_weight.shape
    # layer.weight.data = mask_tensor * layer.weight.data
    layer.weight.data = layer.weight.data.clone().detach().requires_grad_(True) * mask_tensor
```

The code snippet shown above is the body of the `prune_by_percentage` function. First, the percentile is found based on q. Then, the mask is generated. Finally, the mask is applied to the weight. Please refer to the PDF of the Jupyter Notebook for more details.

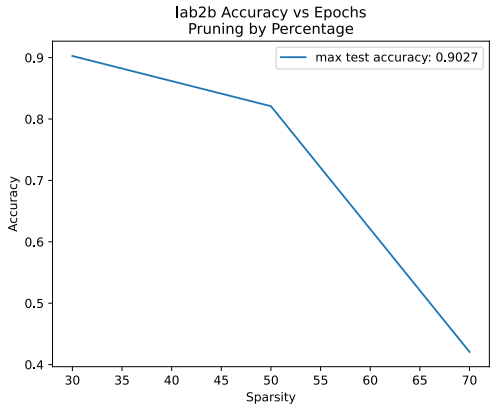


Figure 6: Test accuracy vs pruning percentage q.

The test accuracy values are reported in Figure 6; I assumed that we prune q percent of all weights (instead of keep q percent of all weights un-pruned). For $q = 0.3$, `test_acc = 0.9027`; for $q = 0.5$, `test_acc = 0.8209`; for $q = 0.7$, `test_acc = 0.4207`.

(c) (6 pts) Fill in the `finetune_after_prune` function for pruned model finetuning. Make sure the pruned away elements in previous step are kept as 0 throughout the finetuning process. Finetune the pruned model with $q = 0.7$ for 20 epochs with the provided training pipeline. Report the best accuracy achieved during finetuning. Finish the code for sparsity evaluation to check if the finetuned model preserves the sparsity.

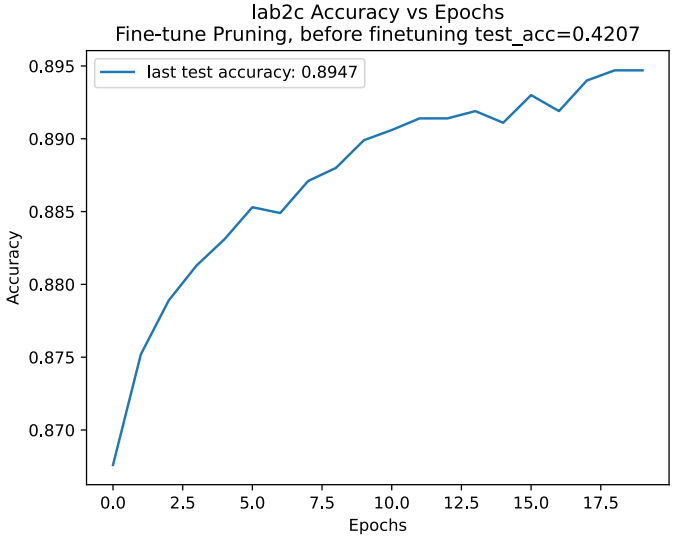


Figure 7: Test accuracy vs epoch for $q = 0.7$.

The best accuracy achieved during fine-tuning is $\boxed{0.8947}$ in the final two epochs (Figure 7).

```
def finetune_after_prune(net, trainloader, criterion, optimizer, prune=True):
...
# Your code here: generate a mask in GPU torch tensor to have 1 for nonzero element and 0 for
# zero element
    layer_weight = layer.weight.detach().cpu().numpy()
    mask_obj = np.ma.masked_equal(x=layer_weight, value=0)  # where it's zero, mask value is True
    weight_mask[name] = torch.tensor(1 - np.ma.getmask(mask_obj).astype(int)).to(device)
...
# Your code here: Use weight_mask to make sure zero elements remains zero
    layer.weight.data = layer.weight.data.clone().detach().requires_grad_(True) * weight_mask[name]
...
```

The mask generation process and the actual masking process in part (c) are very similar to the code that I explained in part (b). To generate a mask, I used the `np.ma.masked_equal` and moved the mask onto the GPU (note: the making function generates `True` values where the input is zero, meaning that I need to invert the condition by computing `1 - mask`). A different mask is generated for each layer. I ensured that the weight elements remained at zero after back-propagation in the second part of the code snippet above.

```
net.load_state_dict(torch.load("net_after_finetune.pt"))
for name,layer in net.named_modules():
    if (isinstance(layer, nn.Conv2d) or isinstance(layer, nn.Linear)) and 'id_mapping' not in name:
        # Your code here:
        ## Convert the weight of "layer" to numpy array
        np_weight = layer.weight.detach().cpu().numpy()
        ## Count number of zeros
        zeros = sum((np_weight == 0).flatten())
        ## Count number of parameters
        total = len(np_weight.flatten())
        # Print sparsity
        print('Sparsity of %s: %g' % (name, zeros/total))
test(net)
```

The code snippet above shows how I made sure that all the layers still had the same number of nonzero elements. I changed the tensor into a numpy array and compted the ratio between the number of zeros and the total length in that numpy array. Please refer to the PDF of the Jupyter Notebook for more details. Please see the next page to confirm that sparsity is preserved.

```
Sparsity of head_conv.0.conv: 0.699074
Sparsity of body_op.0.conv1.0.conv: 0.700087
Sparsity of body_op.0.conv2.0.conv: 0.700087
Sparsity of body_op.1.conv1.0.conv: 0.700087
Sparsity of body_op.1.conv2.0.conv: 0.700087
Sparsity of body_op.2.conv1.0.conv: 0.700087
Sparsity of body_op.2.conv2.0.conv: 0.700087
```

```
Sparsity of body_op.3.conv1.0.conv: 0.69987
Sparsity of body_op.3.conv2.0.conv: 0.699978
Sparsity of body_op.4.conv1.0.conv: 0.699978
Sparsity of body_op.4.conv2.0.conv: 0.699978
Sparsity of body_op.5.conv1.0.conv: 0.699978
Sparsity of body_op.5.conv2.0.conv: 0.699978
Sparsity of body_op.6.conv1.0.conv: 0.699978
Sparsity of body_op.6.conv2.0.conv: 0.700005
Sparsity of body_op.7.conv1.0.conv: 0.700005
Sparsity of body_op.7.conv2.0.conv: 0.700005
Sparsity of body_op.8.conv1.0.conv: 0.700005
Sparsity of body_op.8.conv2.0.conv: 0.700005
Sparsity of final_fc.linear: 0.7
```

(d) (5 pts) Implement iterative pruning. Instead of applying single step pruning before finetuning, try to iteratively increase the sparsity of the model before each epoch of finetuning. Linearly increase the pruning percentage for 10 epochs until reaching 70% in the final epoch (prune by $(7 \times e)\%$ before epoch epoch) then continue finetune for 10 epochs. Pruned weight can be recovered during the iterative pruning process before the final pruning step. Compare performance with (c).

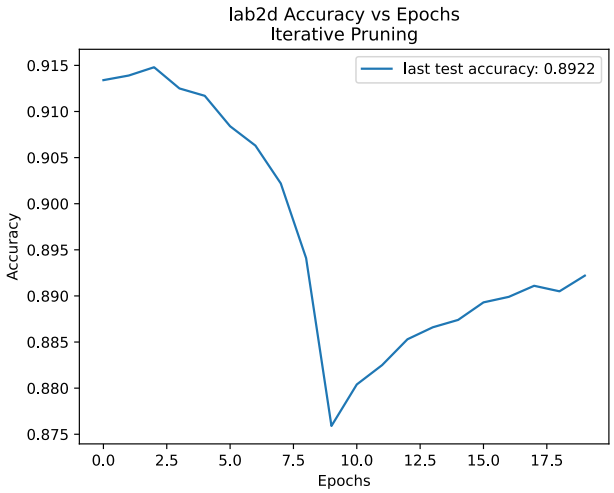


Figure 8: Test accuracy vs epoch, iterative pruning.

```
if epoch < 10:
    for name,layer in net.named_modules():
        if (isinstance(layer, nn.Conv2d) or isinstance(layer, nn.Linear)) and 'id_mapping' not in name:
            # Increase model sparsity
            q = (epoch + 1) * 7
            # Linearly increase the pruning percentage for 10 epochs until reaching 70% in the final epoch
            prune_by_percentage(layer, q=q)
if epoch < 9:
    finetune_after_prune(net, trainloader, criterion, optimizer, prune=False)
else: # starts from epoch==9, where q = 70%
    finetune_after_prune(net, trainloader, criterion, optimizer)
```

The code snippet shown above shows the changes to `q` for the first 10 epochs.

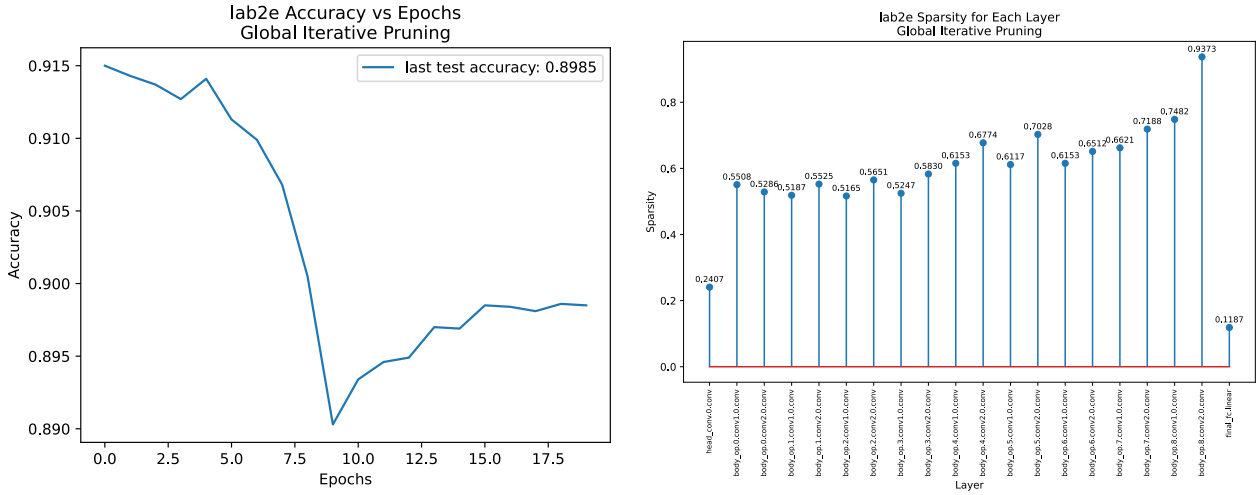
Compare performance with (c):

The performance using iterative pruning (Figure 8) is similar to the performance in (c) (Figure 7). The highest accuracy in part (c) is around the same as the accuracy using iterative pruning.

The weights that were zero before every epoch in part (c) stay zero after the training step, while the weights that were zero before every epoch in iterative finetuning aren't necessarily zero after each training step from epoch 0 to epoch 8 (since `prune=False` in `finetune_after_prune(...)` before epoch 9). The model in (c) isn't able to change the mask while iterative tuning allows the model to learn the optimal location of the nonzero weights. However, the ability to learn the location of the nonzero values doesn't seem to help the model's performance after 20 epochs.

The main effect of iterative pruning is that the accuracy initially decreases as q increases and then increases when q levels off at 70%. Since the model has a different mask for each epoch from epoch 0 to 8, the location of the zero weights are unstable and the model isn't able to learn well in the first few epochs. However, we see that the model's testing accuracy in epoch 9 (70% pruning, iterative pruning) is higher than the model's testing accuracy in epoch 0 (same percentage pruned, part (c)). This suggests that the model in (d) learned a bit when undergoing gradual iterative pruning. The model in (d) not having enough epochs to converge could explain the slightly lower final accuracy compared to (c).

(e) (6 pts) Perform magnitude-based global iterative pruning. Previously we set the pruning threshold of each layer following the weight distribution of the layer and prune all layers to the same sparsity. This will constrain the flexibility in the final sparsity pattern across layers. In this question, fill in the `global_prune_by_percentage` function to perform a global ranking of the weight magnitude from all the layers, and determine a single pruning threshold by percentage for all the layers. Repeat iterative pruning to 70% sparsity, and report final accuracy and the percentage of zeros in each layer.



(a) Test accuracy vs epoch.

(b) Sparsity of each layer.

Figure 9: Lab2 part e, global iterative pruning.

```
def global_prune_by_percentage(net, q=70.0):
    # A list to gather all the weights
    flattened_weights = []
    # Find global pruning threshold
    for name,layer in net.named_modules():
        if (isinstance(layer, nn.Conv2d) or isinstance(layer, nn.Linear)) and 'id_mapping' not in name:
            # Convert weight to numpy
            # Flatten the weight and append to flattened_weights
            flattened_weights.append(layer.weight.detach().cpu().numpy().flatten())

    # Concate all weights into a np array
    flattened_weights = np.concatenate(flattened_weights)
    # Find global pruning threshold
    threshold = np.percentile(np.abs(flattened_weights.flatten()), q)

    # Apply pruning threshold to all layers
    for name,layer in net.named_modules():
        if (isinstance(layer, nn.Conv2d) or isinstance(layer, nn.Linear)) and 'id_mapping' not in name:
            # Convert weight to numpy
            layer_weight = layer.weight.detach().cpu().numpy()

            # Generate a binary mask same shape as weight to decide which element to prune
            masked_obj = np.ma.masked_greater_equal(x=np.abs(layer_weight), value=threshold, copy=True)
            mask_int = np.ma.getmask(masked_obj).astype(int)
            # Convert mask to torch tensor and put on GPU
            mask_tensor = torch.tensor(mask_int).to(device)
            # Multiply the weight by mask to perform pruning
            layer.weight.data = layer.weight.data.clone().detach().requires_grad_(True) * mask_tensor
```

The above code snippet shows the global pruning function. First, I get a list of all the weights and find the threshold based on the percentile q. Then, I create a mask for each layer and apply the mask to the weights of that layer.

Global iterative pruning at $q = 70\%$ achieves test accuracy of 0.8985 in the last epoch according to Figure 9a. The sparsity for each layer is presented in Figure 9b. Please refer to the PDF of the Jupyter Notebook for more details.

4 Lab 3: Fixed-point quantization and finetuning (25 pts)

(a) (10 pts) As is mentioned in lecture 15, to train a quantized model we need to use floating-point weight as trainable variable while use a straight-through estimator (STE) in forward and backward pass to convert the weight into quantized value. Intuitively, the forward pass of STE converts a float weight into fixed-point, while the backward pass passes the gradient straightly through the quantizer to the float weight.
To start with, implement the STE forward function in `FP_layers.py`, so that it serves as a linear quantizer with dynamic scaling, as introduced on page 9 of lecture 15. Please follow the comments in the code to figure out the expected functionality of each line. **Take a screenshot** of the finished STE class and paste it into the report. Submission of the `FP_layers.py` file is not required.

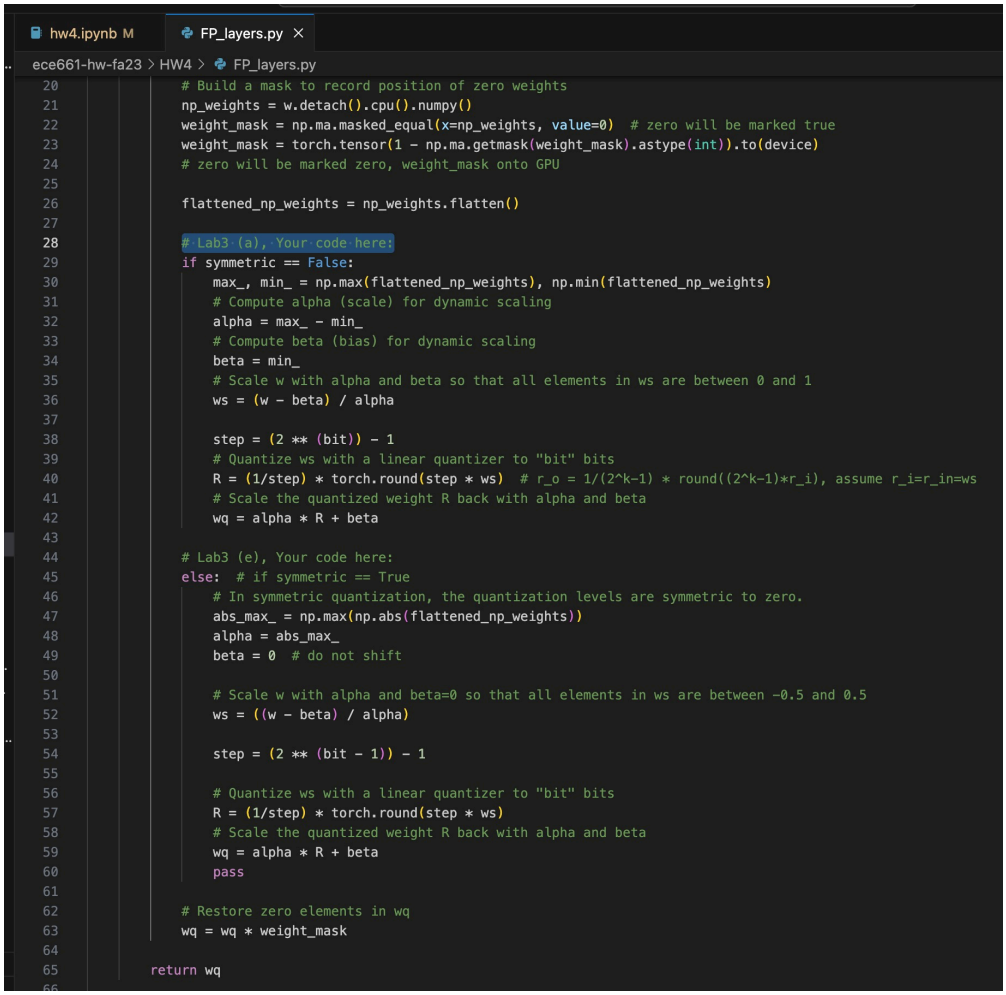


Figure 10: Partial code, `FP_layers.py`.

The `symmetric == False` case implements the algorithm shown on Slide 9, Lecture 15. First, I compute `alpha` and `beta`. Then, I compute the shifted and scaled weight `ws` whose values should all be between $\{0, 1\}$. The scaled weight is then quantised on line 40 and the original values are recovered on line 42. Finally, the mask is applied on line 63.
The code block under the `else` condition completes the requirements for lab 3 (e).

(b) (2 pts) In `hw4.ipynb`, load pretrained ResNet-20 model, **report the accuracy of the floating-point pretrained model**. Then set `Nbits` in the first line of block 4 to 6, 5, 4, 3, and 2 respectively, run it and **report the test accuracy** you got.
The floating point pretrained model has the following accuracy:

```
before quantisation: Test accuracy=0.9150
```

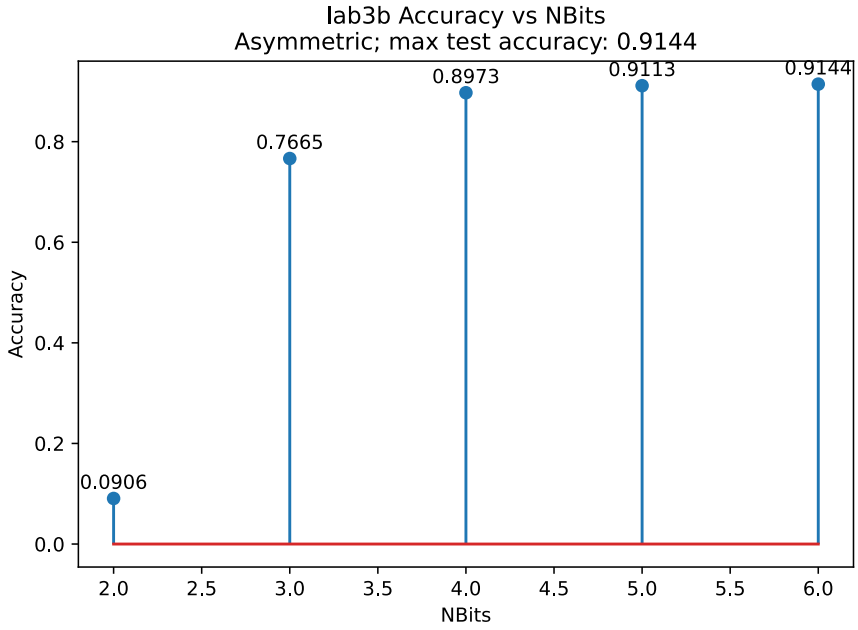


Figure 11: Test accuracy vs `Nbits`.

Please refer to Figure 11 for the test accuracy quantisation. The test accuracy becomes lower as the number of bits used in the quantisation decreases. This is expected because when `Nbits` becomes smaller, the number of distinct values in the weights become smaller. Smaller `Nbits` values mean that the model is less likely to record the original distribution of the weights accurately, especially for weights that are of high magnitudes.

(c) (5 pts) With `Nbits` set to 4, 3, and 2, run code blocks 4 and 5 to finetune the quantised model for 20 epochs. You do not need to change other parameter in the finetune function. For each precision, report the highest testing accuracy you get during finetuning.

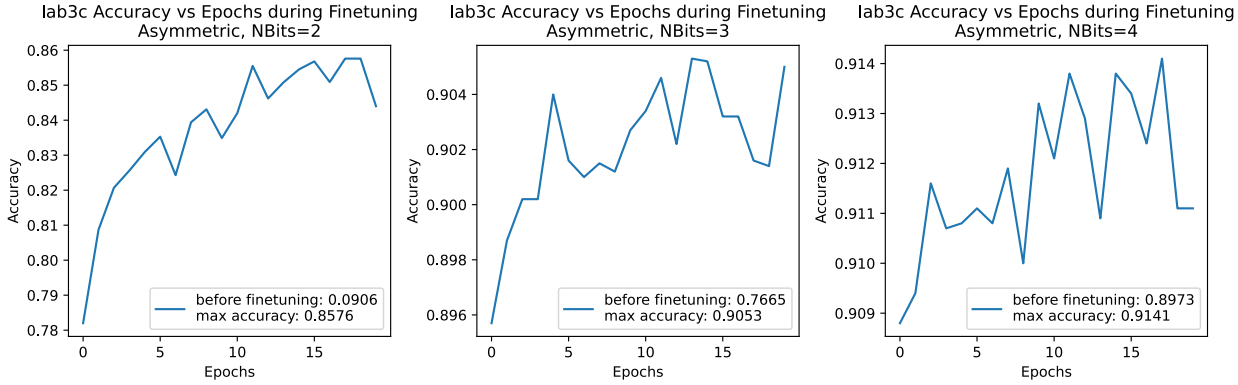


Figure 12: Test accuracy vs Epoch for `Nbits` $\in \{4, 3, 2\}$.

The highest test accuracy for each quantisation (shown in Figure 12) is higher than the test accuracy before fine-tuning. Even after a one epoch the testing accuracy becomes higher than the testing accuracy before finetuning for `Nbits == 2` and `Nbits == 3`, which means that the quantised model benefits a great deal from finetuning when `Nbits` is small.

Comment on the relationship between precision and accuracy, and on the effectiveness of finetuning.
As the precision increases (higher value for `Nbits`), the highest test accuracy increases. With more bits, the model is able to change more flexibly during the training process and thus reach a state where the loss is lower. Too much information is lost about the model when the number of bits used to represent weight values is too low, causing the model to be less accurate. There's also too much constraints on what the values of each weight take on, meaning that convolution and fully-connected layers aren't fine-grained enough.

The absolute difference between the final test accuracy and the test accuracy before finetuning decreases as `Nbits` increases. When `Nbits` is large, the model already performs quite well before finetuning occurs, meaning that there's limited room for further improvement. When `Nbits` is small (`Nbits == 2` for example), there's a lot of room for the accuracy to increase given that the initial accuracy before finetuning with `Nbits == 2` is so low. **So, we can say that finetuning is less effective when `Nbits` is large.**

```
Nbits_arr_c = [2, 3, 4]
test_acc_lab3c_finetune = {}
# test accuracy for each finetuning as fn of epoch
final_test_acc_lab3c = {}  # test acc after finetuning for each Nbit
n_epochs = 20
test_acc_lab3c_b4_finetune = {} # test acc before finetuning

for iii, nb in enumerate(Nbits_arr_c):
    # Define quantized model and load weight
    Nbits = nb #Change this value to finish (b) and (c)

    net = ResNetCIFAR(num_layers=20, Nbits=Nbits)
    net = net.to(device)
    net.load_state_dict(torch.load("pretrained_model.pt"))
    test_acc_lab3c_b4_finetune[Nbits] = test(net)

    # Quantized model finetuning
    test_acc_lab3c_finetune[Nbits] = np.asarray(
        finetune(net, epochs=n_epochs, batch_size=256, lr=0.002, reg=1e-4))
    # finetune() returns list of val_accuracy values because I edited it
    # Load the model with best accuracy
    net.load_state_dict(torch.load("quantized_net_after_finetune.pt"))
    final_test_acc_lab3c[Nbits] = test(net)
```

The code snippet above shows the process for fine-tuning the quantised model. In each iteration, `Nbits` is changed, the model is re-created & re-loaded with the pretrained model, and its accuracy is recorded before finetuning. The best model that resulted from finetuning is loaded and its test accuracy is recorded.

(d) (4 pts) In practice, we want to apply both pruning and quantization on the DNN model. Here we explore how pruning will affect quantization performance. Please load the checkpoint of the 70% sparsity model with the best accuracy from Lab 2, repeat the process in (c), and report the accuracy before and after finetuning.

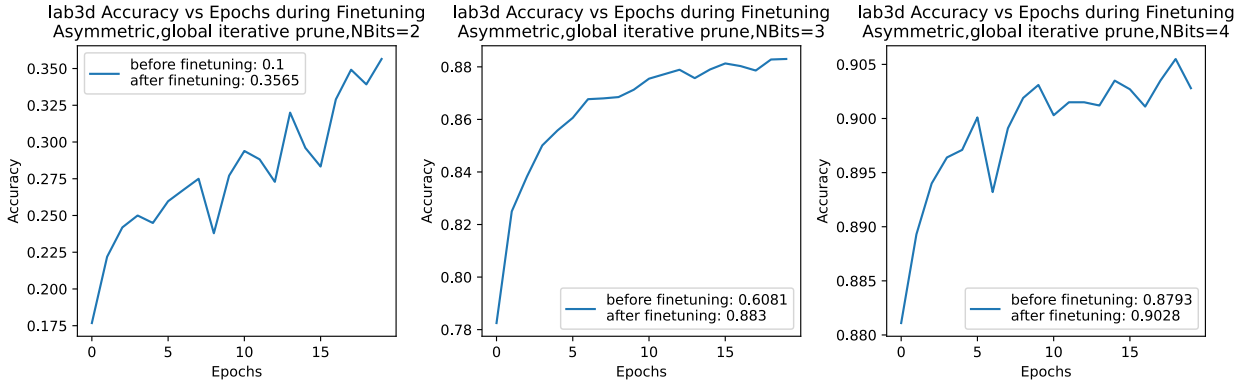


Figure 13: Test accuracy vs Epoch for `Nbits` $\in \{4, 3, 2\}$ with model trained using global iterative pruning.

The test accuracy values before and after finetuning are reported in Figure 13. Note that the accuracy before finetuning is reported for the quantised version of the best global iteratively pruned model.

Discuss your observations comparing to (c)'s results:
Before finetuning, the model in (d) performs worse than the models in (c). This makes sense because the model in (d) loses more information than the model in (c) does before any finetuning occurs. The test accuracy values after finetuning for the model in (d) is also lower than the corresponding values for the model in (c). This also makes sense because the model in part (d) started off in a less accurate state compared to the model in part (c) since the model is quantised *and* the model was pruned at the beginning. Although the `finetune(...)` function does not appear to force zero values to remain at zero after each training step (this means that the model will become less sparse over time), the model that we started with in part (d) is imprecise enough that it takes more time to train the model.
These effects are especially strong with `Nbits == 2` in part (d). Although the corresponding model in part (c) was able to achieve an accuracy as high as 0.86 during finetuning, part (d)'s model with `Nbits == 2` only achieves an accuracy of 0.36 after finetuning.

```
nbits_lab3d = [2, 3, 4]
test_acc_lab3d_b4_finetune = {}
test_acc_arrays_finetune = {}
test_acc_lab3d_after_finetune = {}
n_epochs = 20

for iii, nb in enumerate(nbits_lab3d):
    # Define quantized model and load weight
    Nbits = nb #Change this value to finish (d)

    net = ResNetCIFAR(num_layers=20, Nbits=Nbits)
    net = net.to(device)
    net.load_state_dict(torch.load("net_after_global_iterative_prune.pt"))
    test_acc_lab3d_b4_finetune[Nbits] = test(net)

    # Quantized model finetuning
    test_acc_arrays_finetune[Nbits] = np.array(
        finetune(net, epochs=n_epochs, batch_size=256, lr=0.002, reg=1e-4))

    # Do not load the model with best accuracy since we want the last acc value.
    # net.load_state_dict(torch.load("quantized_net_after_finetune.pt"))
    test_acc_lab3d_after_finetune[Nbits] = test(net)
```

The above code snippet shows the finetuning process for the model that came about using global iterative prune. In each iteration of the loop, the net is re-loaded, tested, and then fine-tuned. Different from the previous page (lab 3 part (c)), the net is tested again without loading the net with the best result. This records the accuracy both before and after finetuning.

(e) (4 pts) Symmetric quantization is a commonly used and hardware-friendly quantization approach. In symmetric quantization, the quantization levels are symmetric to zero. Implement symmetric quantization in the STE class and repeat the process in (b). Compare and analyze the performance of symmetric quantization and asymmetric quantization.

Please refer to the bottom half of Figure 10 for details on how I implemented quantisation where `symmetric == True`. I computed the quantised weights using this formula:

$$w_q = w_{\text{mask}} \odot \left[\frac{1}{2^{\text{bit}-1} - 1} \cdot \text{round}\left[(2^{\text{bit}-1} - 1) \cdot w \div \max(\text{abs}(w)) \right] \right] \cdot \max(\text{abs}(w))$$

Figure 14 shows that the accuracy of asymmetric quantisation is higher than that of symmetric quantisation.

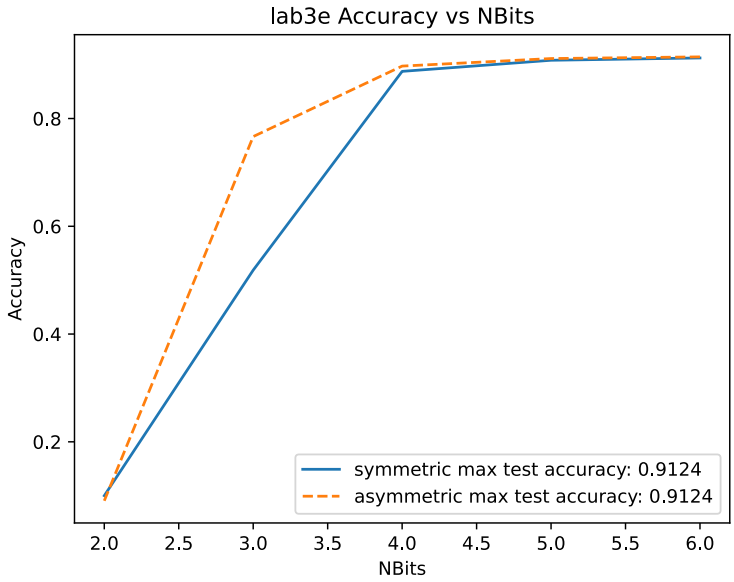


Figure 14: Final test accuracy `Nbits` $\in \{4, 3, 2\}$, comparing asymmetric and symmetric pruning.

The range of values in the quantisation for the symmetric case is larger than or equal to that of the asymmetric case since $\alpha = \max |w|$ in the symmetric case ($\alpha = \max w - \min w$ in the asymmetric case). This means that the numerical differences between adjacent quantisation levels are larger. Since symmetric quantisation results in quantisation levels that are spaced further apart, the weight values are mapped less finely when using symmetric quantisation. When a larger-than-necessary range is used in symmetric quantisation, it results in higher loss of information compared to the asymmetric case.
When `Nbits` $= 3$, asymmetric quantisation has just enough bits to capture the distribution of high-magnitude weights and predict with passable accuracy, while the symmetric quantisation loses just enough precision to predict less accurately. With `Nbits` $= 2$, the number of bits in both the asymmetric and symmetric cases are too low to capture the higher-magnitude weights, while with `Nbits` $\in \{4, 5, 6\}$, the number of bits is high enough that the small-ish loss in accuracy (loss in accuracy compares symmetric and asymmetric quantisation) caused by using symmetric quantisation doesn't decrease the accuracy significantly.

tl;dr: The higher information loss in the symmetric case means that the distributions of the high-magnitude weights are captured less well, causing the performance to be lower.
The piece of text below prints the test accuracy for symmetric quantisation.

```
Nbits=2, Test accuracy=0.1000
Nbits=3, Test accuracy=0.5186
Nbits=4, Test accuracy=0.8875
Nbits=5, Test accuracy=0.9081
Nbits=6, Test accuracy=0.9124
```

lab1

November 8, 2023

[294]:
```
import sys
print(sys.executable)
```

```
/Users/zeyuli/anaconda3/envs/ECE661_HW/bin/python
```

[295]:
```
import numpy as np
import matplotlib.pyplot as plt
from typing import Tuple

x1, y1 = np.array([ 1, -2, -1, -1,  1]), 7
x2, y2 = np.array([ 2, -1,  2,  0, -2]), 1
x3, y3 = np.array([-1,  0,  2,  2,  1]), 1

x_input = np.array([x1, x2, x3]).reshape(3, 5)
y_input = np.array([y1, y2, y3])

w0 = np.zeros(5).reshape(-1, 1)
mu = 0.02
n_epochs = 200
print(x_input, y_input)

LEGEND_size = 7
TITLE_size = 12
AXLABEL_size = 9
TICK_size = 8
```

```
[[ 1 -2 -1 -1  1]
 [ 2 -1  2  0 -2]
 [-1  0  2  2  1]] [7 1 1]
```

[296]:
```
def my_relu(a):
    return np.abs(a) * (a > 0)


def l1_norm_trimmed_update(in_x, in_y, w_prev, l1_reg_param, lr, thresh):
    """
    thresh: number of elements to keep unchanged when performing proximal l1
    """
```

```
    ascend_idx = np.argsort(np.abs(w_prev.flatten()))
    weight_l1_proximal = l1_norm_proximal_update(in_x, in_y, w_prev,␣
 ↪l1_reg_param, lr,
                                                 which_weights=ascend_idx[0:
 ↪w_prev.shape[0] - thresh])
    return weight_l1_proximal


def l1_norm_proximal_update(in_x, in_y, w_prev, l1_reg_param, lr,␣
 ↪which_weights=None):
    """
    Computes weight using proximal gradient descent (l1-norm). threshold is␣
 ↪l1_reg_param*lr
    Args:
        in_x: x values (data)
        in_y: y values (target)
        w_prev: previous weight
        l1_reg_param: l1 regularisation (lambda)
        lr: learning rate, must be not None.
        which_weights: indices of which weights for which proximal update is␣
 ↪applied
    Note: threshold = lr * lambda should be in set{0.004, 0.01, 0.02, 0.04}
    Returns: updated weight
    """
    if which_weights is None:
        which_weights = np.array(range(len(w_prev.flatten())))
        # which_weights = [0,1,2,3,4] if w_prev.shape == (5, 1)
    weight_tilde = no_regularisation_update(in_x, in_y, w_prev, lr=lr) # main␣
 ↪objective
    weight_ret = np.copy(weight_tilde)

    threshold = lr * l1_reg_param
    assert threshold >= 0

    for iii in which_weights:  # only updates the weights specified in␣
 ↪which_weights
        this_weight = weight_tilde[iii, 0]
        weight_ret[iii, 0] = my_relu(np.abs(this_weight) - threshold) * np.
 ↪sign(this_weight)
    return weight_ret


def l1_norm_update(in_x, in_y, w_prev, l1_reg_param, lr):
    """
    Args:
        in_x: x values (data)
```

```
        in_y: y values (target)
        w_prev: previous weight
        l1_reg_param: l1 regularisation (lambda)
        lr: learning rate, must be not None
    Returns: updated weight
    """
    grad = no_regularisation_update(in_x, in_y, w_prev, l1_reg_param, lr=None)␣
↪+ \
           l1_reg_param * np.sign(w_prev)
    assert grad.shape == w_prev.shape
    return w_prev - lr * grad


def no_regularisation_update(in_x, in_y, w_prev, l1_reg_param=None, lr=None):
    """
    Args:
        in_x: x values (data)
        in_y: y values (target)
        w_prev: previous weight
        l1_reg_param: dummy
        lr: if None, it means some fn other than run_grad_descent called it; if␣
↪run_grad_descent called it,
        must be (not None)
    Returns: returns updated weight if lr is not None; returns gradient if lr␣
↪is None
    """
    n_r, n_c = in_x.shape
    grad = 0
    for inner in range(n_r):
            grad += 2 * (np.dot(in_x[inner, :], w_prev) - in_y[inner]) *␣
↪in_x[inner].reshape(-1, 1)
    if lr is not None:
        return w_prev - lr * grad
    else:
        return grad


def run_grad_descent(w_init, lr, in_x, in_y, epochs, w0_prune=False,␣
↪thresh=None, l1_reg_param=None,
                     grad_fn=no_regularisation_update) -> Tuple[np.array, np.
↪array, np.array]:
    """
    Args:
        w_init: w0 = np.zeros(5)
        lr: mu=0.02
        in_x: x1, x2, x3
```

```
        in_y: y1, y2, y3
        epochs: 200
        w0_prune: prune or not (lab 1 c). should be False in all except for lab␣
↪1c
        thresh: threshold for pruning or trimmed l1. should be True only in lab␣
↪1 c, lab 1 f
        l1_reg_param: lambda, used in update weight
        grad_fn: function for updating weight
    Returns: optimal weight (np.array), loss for each epoch (np.array), and␣
↪w_arr matrix to
    record the progression of each weight through time
    """
    w_prev = w_init
    l_arr = []
    w_arr = np.zeros((len(w_init), epochs))
    n_r, n_c = in_x.shape
    for i in range(epochs):
        if not w0_prune and thresh is not None:  # only for lab 1 f, l1-trimmed
            w_curr = grad_fn(in_x, in_y, w_prev, l1_reg_param, lr=lr,␣
↪thresh=thresh)
        else:
            w_curr = grad_fn(in_x, in_y, w_prev, l1_reg_param, lr=lr)

        if w0_prune and thresh is not None:  # only for lab 1 c, pruning
            w_curr_sorted_idx = np.argsort(np.abs(w_curr.flatten()))  # sort␣
↪ascending
            for inner in range(len(w_init) - thresh):
                # len(w_init) - thresh = 5 - 2 = 3, so three smallest weights␣
↪(absolute value) are
                # forced to zero
                w_curr[w_curr_sorted_idx[inner], 0] = 0  # zero-ing out␣
↪smallest elements
        l_curr = 0
        for inner in range(n_r):
            l_curr += (np.dot(in_x[inner, :], w_curr) - in_y[inner]) ** 2
        l_arr.append(l_curr)
        w_arr[:, i] = w_curr.flatten()
        w_prev = w_curr
    return w_prev, np.array(l_arr), w_arr


def plot_log_loss_vs_n_iters(num_epochs, loss_array, save_name_, save=False,
                             plt_axis=None):
    if plt_axis is None:
        fig, ax = plt.subplots(1, 1)
    else:
```

```
        fig, ax = None, plt_axis
    ax.semilogy(range(num_epochs), loss_array, label='Lowest loss=%g' % np.
 ↪min(loss_array))
    ax.set_xlabel('Number of Iterations', fontsize=AXLABEL_size)
    ax.set_ylabel('L (log10 scale)', fontsize=AXLABEL_size)
    ax.set_title('Loss vs Number of Iterations', fontsize=TITLE_size)
    ax.tick_params(axis='both', labelsize=TICK_size)
    ax.legend(loc='best', fontsize=LEGEND_size)
    if fig is not None:
        fig.tight_layout()
    if save:
        plt.savefig('%s.pdf' % save_name_, dpi=700, bbox_inches='tight')


def plot_weight_val_vs_n_iters(num_epochs, w_init, final_weights, weight_array,␣
 ↪save_name_, save=False,
                               plt_axis=None):
    if plt_axis is None:
        fig, ax = plt.subplots(1, 1)
    else:
        fig, ax = None, plt_axis
    line_styles = ['-', ':', '--', '-.', (0, (1, 1))]
    for iii in range(len(w_init)):
        ax.plot(range(num_epochs), weight_array[iii, :],␣
 ↪linestyle=line_styles[iii],
                linewidth=1., label='final value %.5f' % final_weights[iii])
    ax.legend(loc='best', fontsize=LEGEND_size)
    ax.tick_params(axis='both', labelsize=TICK_size)
    ax.set_xlabel('Number of Iterations', fontsize=AXLABEL_size)
    ax.set_ylabel('Value of Weights', fontsize=AXLABEL_size)
    ax.set_title('Weight Values vs Number of Iterations', fontsize=TITLE_size)
    if fig is not None:
        fig.tight_layout()
    if save:
        plt.savefig('%s.pdf' % save_name_, dpi=700, bbox_inches='tight')


def plot_many_w_vals_or_loss_vs_n_iters(num_epochs, w_init, l1_reg_params,␣
 ↪save_name_, save=False,
                                       loss_arr_lst=None, final_weights_list=None,␣
 ↪weight_array_list=None):
    """
    Args:
        num_epochs: number of iterations (200)
        w_init: initial weights
        final_weights_list: list of weights achieved by gradient descent by␣
 ↪default None
```

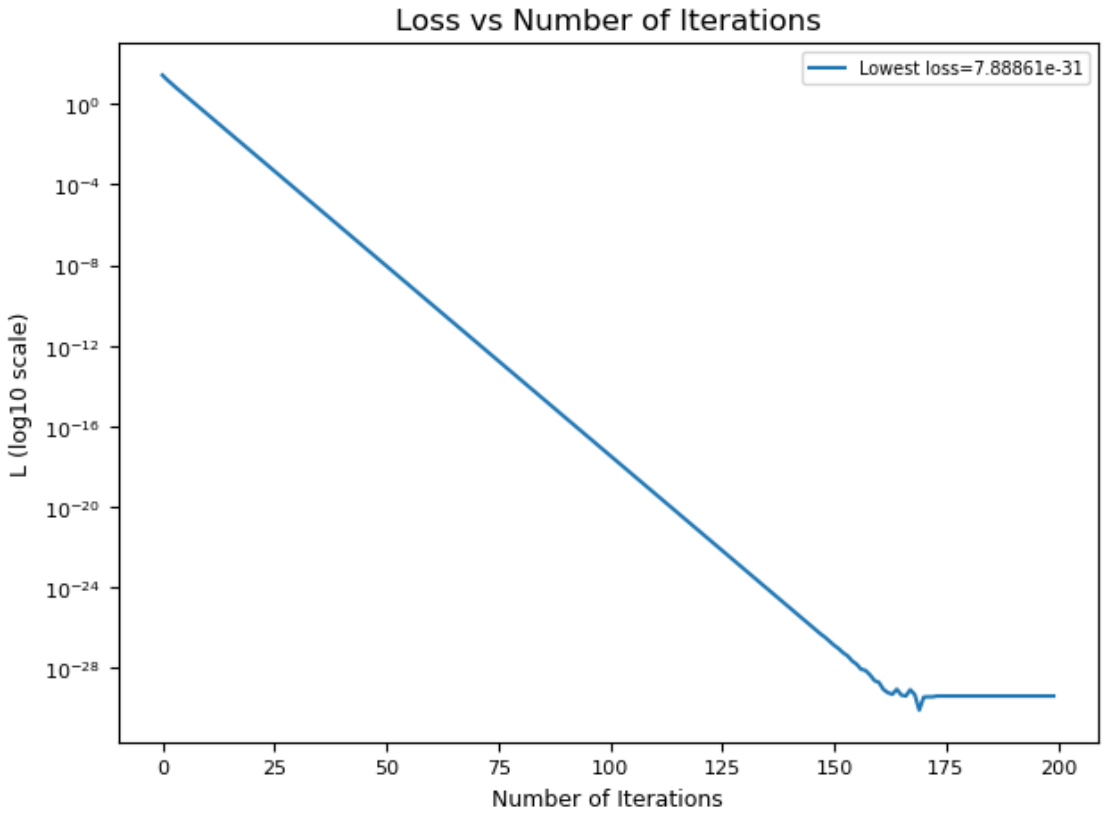
```
        weight_array_list: list of np array that describe weight progression by␣
↪default None
        loss_arr_lst: list of loss arrays (len(loss_arr_lst) corresponds to␣
↪len(l1_reg_params))
        by default, loss_arr_lst=None. If not None,
        l1_reg_params: reg params (lambda)
        save_name_: name of file
        save: save or not
    Either (final_weights_list and weight_array_list are not None) or␣
↪(loss_arr_lst is not None)
    Returns: None
    """
    assert (final_weights_list is not None and weight_array_list is not None)␣
↪or \
        loss_arr_lst is not None, \
        "Either (final_weights_list and weight_array_list are not None) or␣
↪(loss_arr_lst is not None)"
    assert not (final_weights_list is not None and weight_array_list is not␣
↪None and
        loss_arr_lst is not None), "all three are not None."
    fig, ax = plt.subplots(2, 2, figsize=(10, 8))
    ax_flat = ax.flatten()
    for iii, axis in enumerate(ax_flat):
        if loss_arr_lst is None:
            plot_weight_val_vs_n_iters(num_epochs, w_init,␣
↪final_weights=final_weights_list[iii],
                                       weight_array=weight_array_list[iii],␣
↪save_name_=None, save=False,
                                       plt_axis=axis)
        else:
             plot_log_loss_vs_n_iters(num_epochs, loss_array=loss_arr_lst[iii],␣
↪save_name_=None,
                                      save=False, plt_axis=axis)
        ax_flat[iii].set_title("lambda=%g" % l1_reg_params[iii])
    if fig is not None:
        fig.tight_layout()
    if save_name_ is not None and save:
        plt.savefig('%s.pdf' % save_name_, dpi=700, bbox_inches='tight')
    return
```

[297]:
```
""" part b (3 pts) In Python, directly minimize the objective L without any␣
↪sparsity-inducing regularisation/constraint. Plot the value of log(L) vs.␣
↪#steps throughout the training, and use another figure to plot how the value␣
↪of each element in W is changing throughout the training. From your result,␣
↪is W converging to an optimal solution? Is W converging to a sparse solution?
"""
```

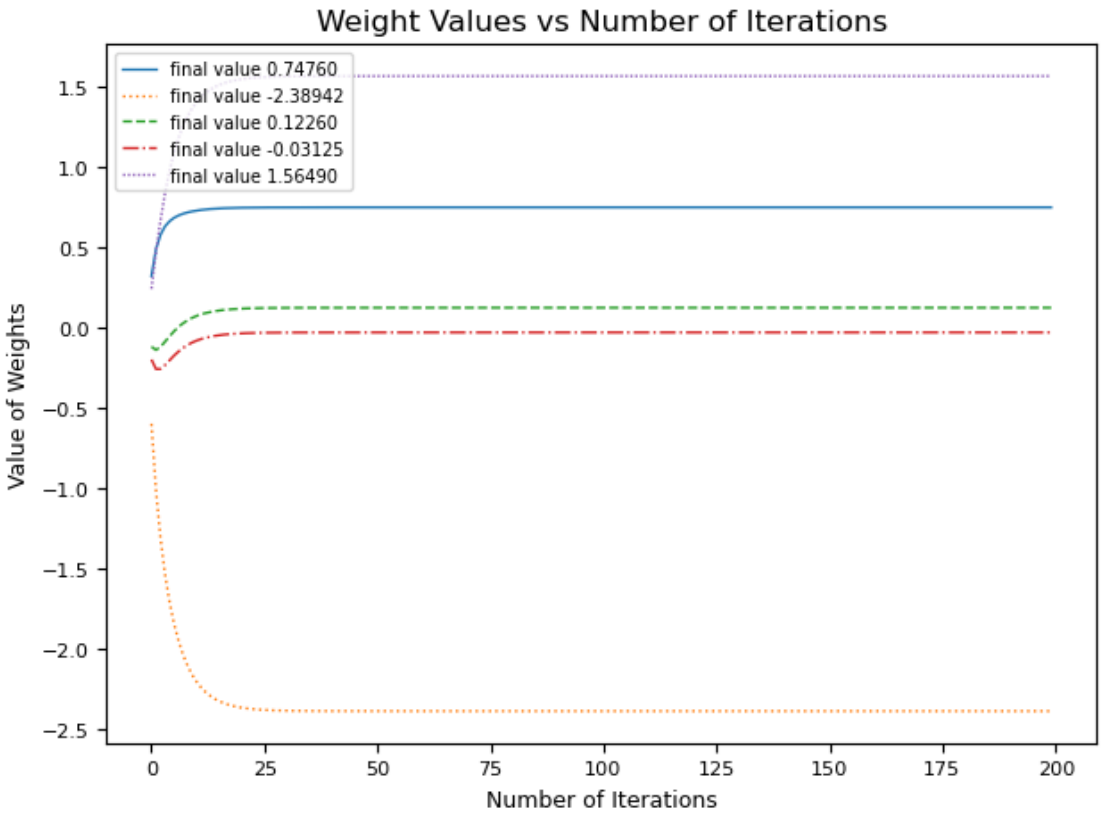
```
weight, loss_arr, weight_arr = run_grad_descent(w0, mu, x_input, y_input,␣
 ↪epochs=n_epochs)
print(weight.shape, loss_arr.shape, weight_arr.shape)
```

```
(5, 1) (200, 1) (5, 200)
```

[298]:
```
# part b
plot_log_loss_vs_n_iters(num_epochs=n_epochs, loss_array=loss_arr,␣
 ↪save_name_='lab1b_loss', save=True)
```

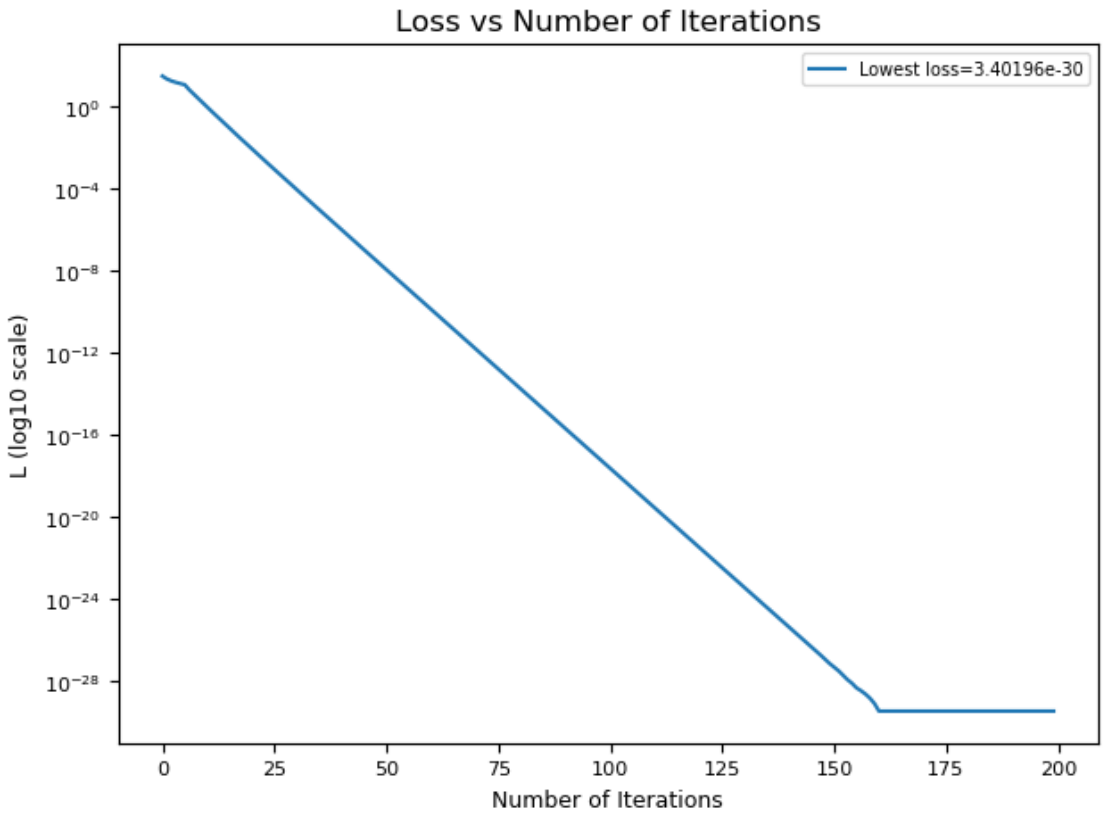


[299]:
```
# part b
plot_weight_val_vs_n_iters(num_epochs=n_epochs, w_init=w0, final_weights=weight,
                           weight_array=weight_arr, save_name_='lab1b_weights',␣
 ↪save=True)
```

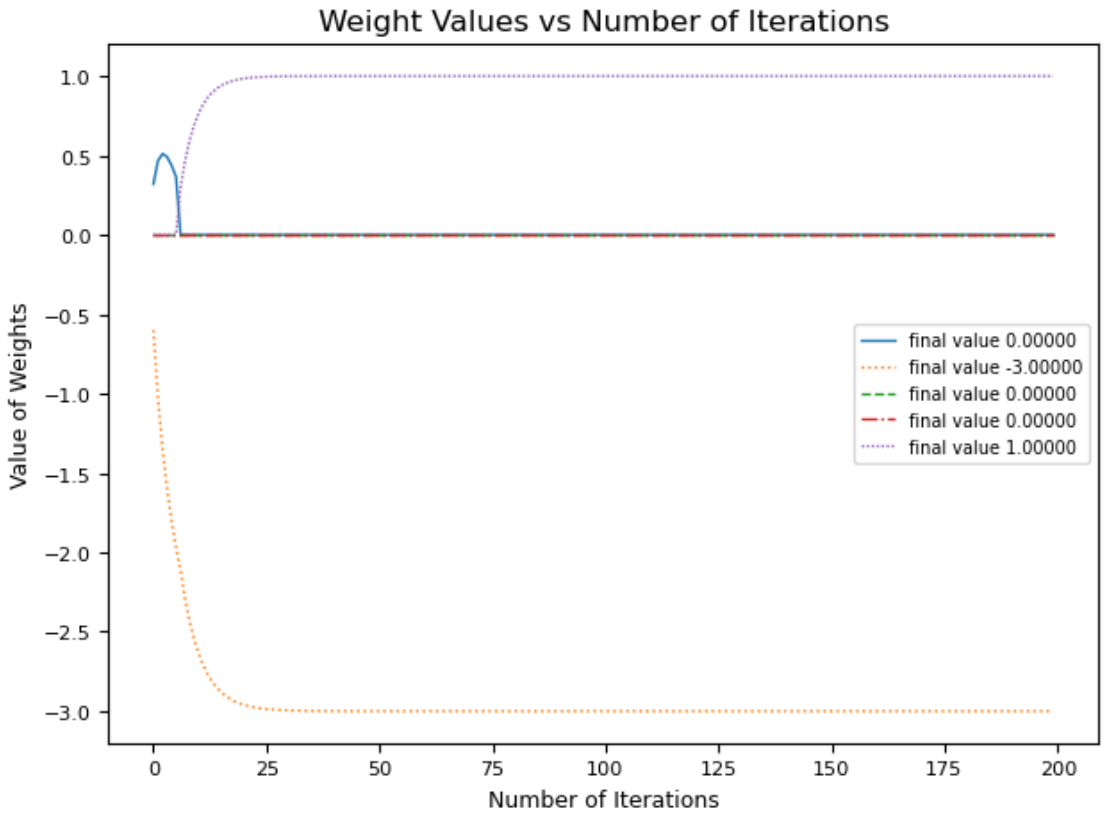


[300]:
```
"""part (c) (6 pts) Since we have the knowledge that the ground-truth weight␣
 ↪should have ||W||0  2, we can apply projected gradient descent to enforce␣
 ↪this sparse constraint. Redo the optimization process in (b), this time␣
 ↪prune the elements in W after every gradient descent step to ensure ||W^l||0␣
 ↪  2. Plot the value of log(L) throughout the training, and use another figure␣
 ↪to plot the value of each element in W in each step. From your result, is W␣
 ↪converging to an optimal solution? Is W converging to a sparse solution?"""
weight_pruned, loss_arr_pruned, weight_arr_pruned = \
    run_grad_descent(w0, mu, x_input, y_input, n_epochs, w0_prune=True,␣
 ↪thresh=2)
```

[301]:
```
plot_log_loss_vs_n_iters(num_epochs=n_epochs, loss_array=loss_arr_pruned,␣
 ↪save_name_='lab1c_loss',
                          save=True)
```



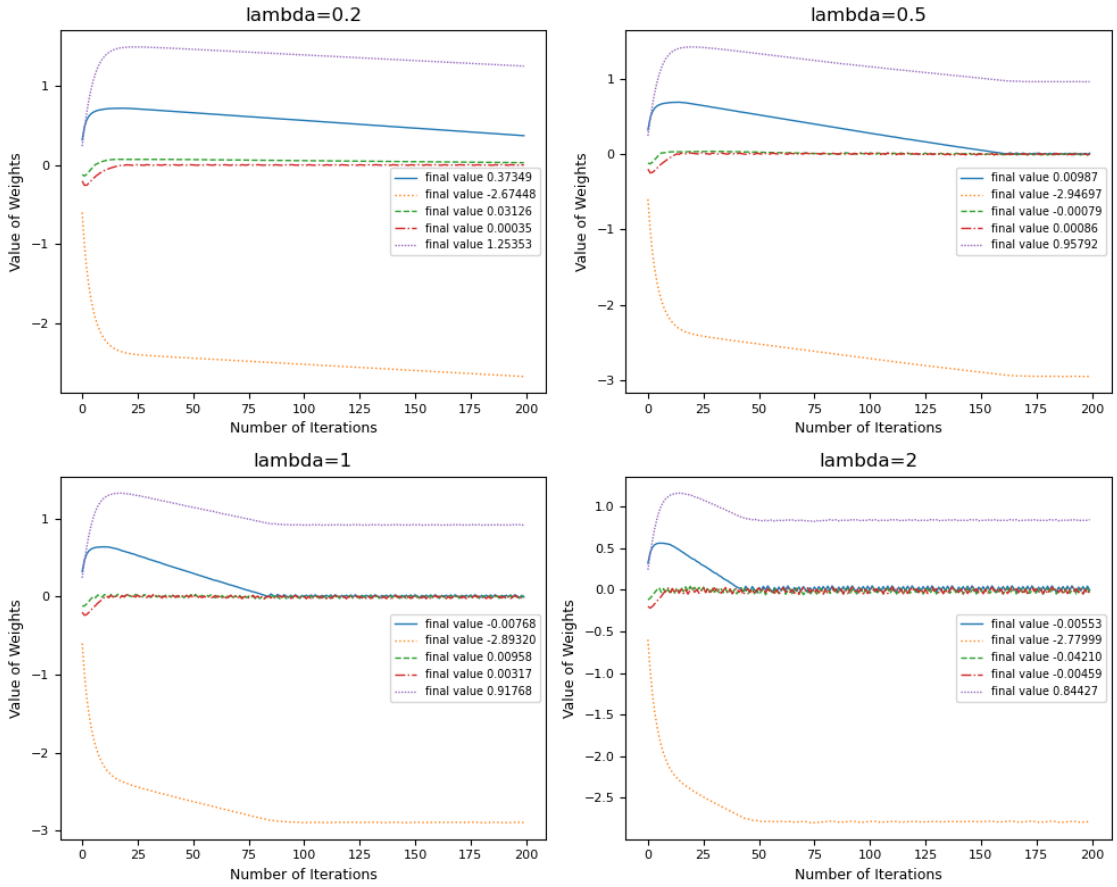
[302]:
```
# part c
plot_weight_val_vs_n_iters(num_epochs=n_epochs, w_init=w0,␣
 ↪final_weights=weight_pruned,
                            weight_array=weight_arr_pruned,␣
 ↪save_name_='lab1c_weights', save=True)
```



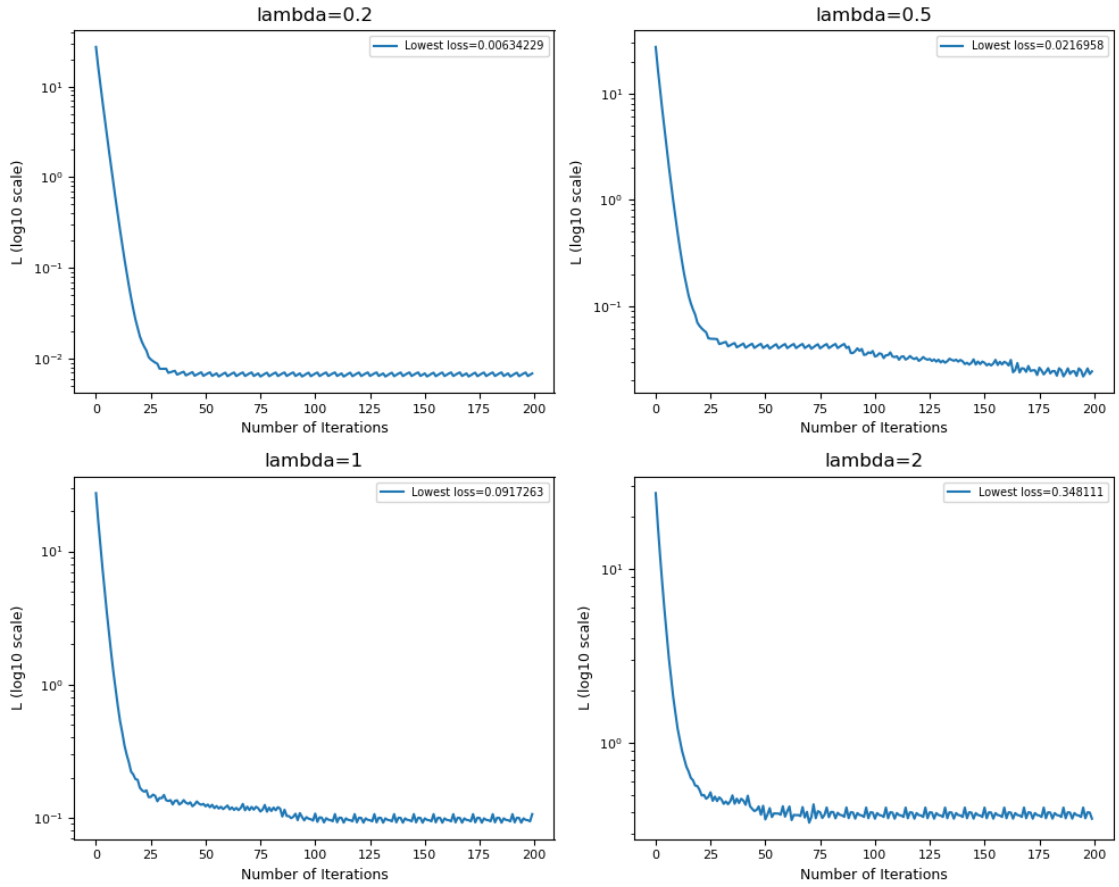
[303]:
```
"""part (d) (5 pts) In this problem we apply l1 regularization to induce the␣
 ↪sparse solution. The minimization objective therefore changes to L +␣
 ↪lambda||W||_1. Please use full-batch gradient descent to minimize this␣
 ↪objective, with lambda = {0.2, 0.5, 1.0, 2.0} respectively. For each case,␣
 ↪plot the value of log(L) throughout the training, and use another figure to␣
 ↪plot the value of each element in W in each step. From your result, comment␣
 ↪on the convergence performance under different lambda."""

lambda_list = [0.2, 0.5, 1.0, 2.0]
weight_l1_list, loss_l1_list, weight_arr_l1_list = [], [], []
for idx, lll in enumerate(lambda_list):
    weight_l1, loss_arr_l1, weight_arr_l1 = run_grad_descent(w0, mu, x_input,␣
 ↪y_input, n_epochs,
                                                             w0_prune=False,␣
 ↪thresh=None,
                                                             ␣
 ↪l1_reg_param=lambda_list[idx],
                                                             ␣
 ↪grad_fn=l1_norm_update)
    weight_l1_list.append(weight_l1)
```

```
    loss_l1_list.append(loss_arr_l1)
    weight_arr_l1_list.append(weight_arr_l1)
```

[304]:
```
plot_many_w_vals_or_loss_vs_n_iters(num_epochs=n_epochs, w_init=w0,␣
↪final_weights_list=weight_l1_list,
                                    weight_array_list=weight_arr_l1_list,␣
↪loss_arr_lst=None,
                                    l1_reg_params=lambda_list,␣
↪save_name_='lab1d_weight', save=True)
```



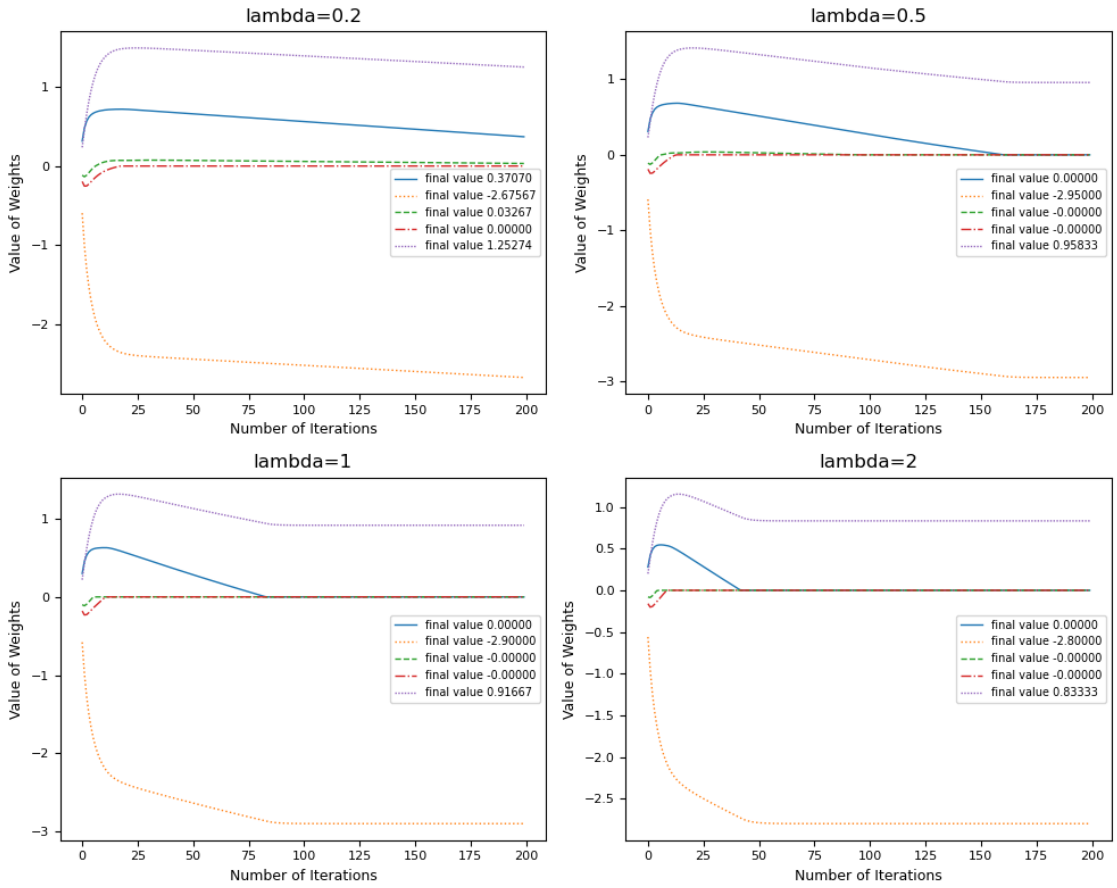
[305]:
```
plot_many_w_vals_or_loss_vs_n_iters(num_epochs=n_epochs, w_init=w0,␣
↪final_weights_list=None,
                                    weight_array_list=None,␣
↪loss_arr_lst=loss_l1_list,
                                    l1_reg_params=lambda_list,␣
↪save_name_='lab1d_loss', save=True)
```



[306]:
```
"""
part (e) (6pts) Here we optimize the same objective as in (d), this time using␣
 ↪proximal gradient update. Recall that the proximal operator of the l1␣
 ↪regulariser is the soft thresholding function. Set the threshold in the soft␣
 ↪thresholding function to {0.004, 0.01, 0.02, 0.04} respectively. Plot the␣
 ↪value of log(L) throughout the training, and use another figure to plot the␣
 ↪value of each element in W in each step. Compare the convergence performance␣
 ↪with the results in (d). (Hint: Optimizing L + lambda||W||_1 using gradient␣
 ↪descent with learning rate mu should correspond to proximal gradient update␣
 ↪with threshold mu*lambda)"""

lambda_list = [0.2, 0.5, 1.0, 2.0]
weight_prox_list, loss_prox_list, weight_arr_prox_list = [], [], []
for idx, lll in enumerate(lambda_list):
    weight_prox, loss_arr_prox, weight_arr_prox = run_grad_descent(w0, mu,␣
 ↪x_input, y_input, n_epochs,
                                                                        ␣

 ↪w0_prune=False, thresh=None,
```

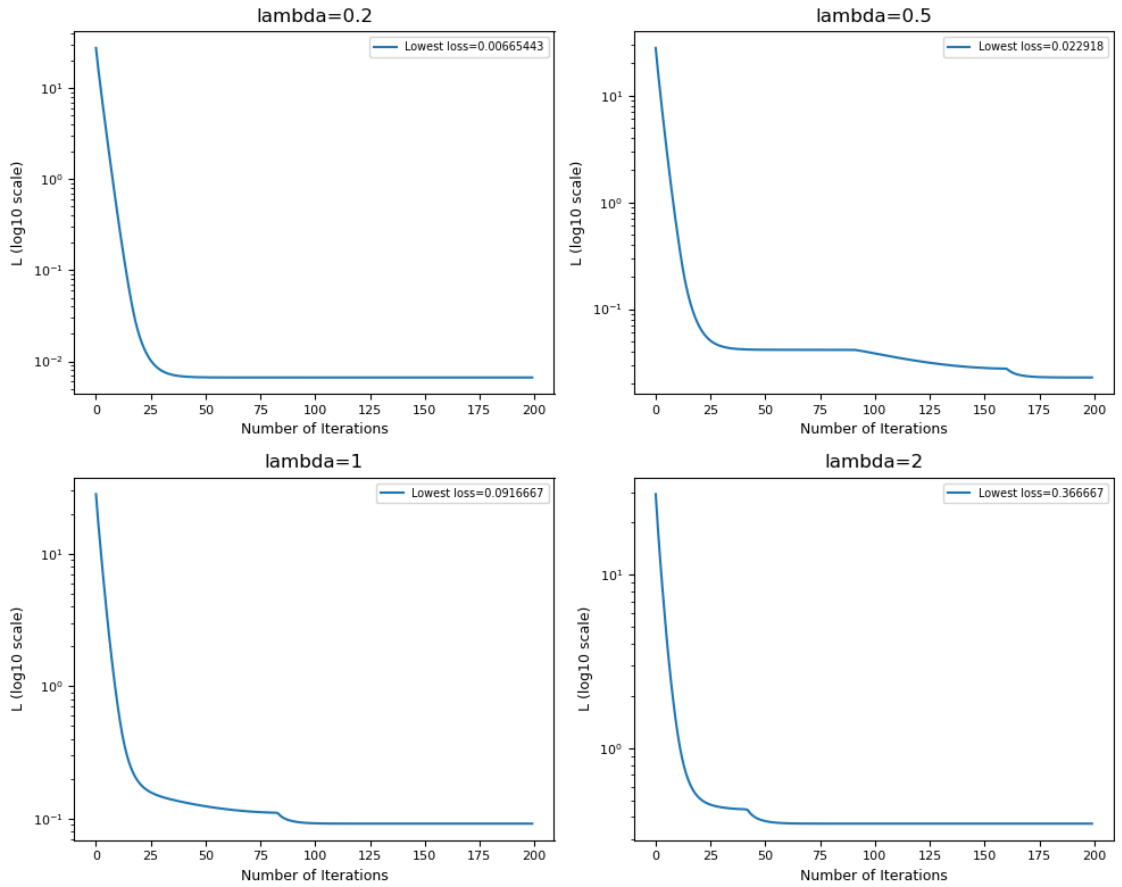
```
                                                                    ␣
↪l1_reg_param=lll,
                                                                    ␣
↪grad_fn=l1_norm_proximal_update)
   weight_prox_list.append(weight_prox)
   loss_prox_list.append(loss_arr_prox)
   weight_arr_prox_list.append(weight_arr_prox)
```

[307]:
```
plot_many_w_vals_or_loss_vs_n_iters(num_epochs=n_epochs, w_init=w0,␣
↪final_weights_list=weight_prox_list,
                                    weight_array_list=weight_arr_prox_list,␣
↪loss_arr_lst=None,
                                    l1_reg_params=lambda_list,␣
↪save_name_='lab1e_weight', save=True)
```



[308]:
```
plot_many_w_vals_or_loss_vs_n_iters(num_epochs=n_epochs, w_init=w0,␣
↪final_weights_list=None,
                                    weight_array_list=None,␣
↪loss_arr_lst=loss_prox_list,
```

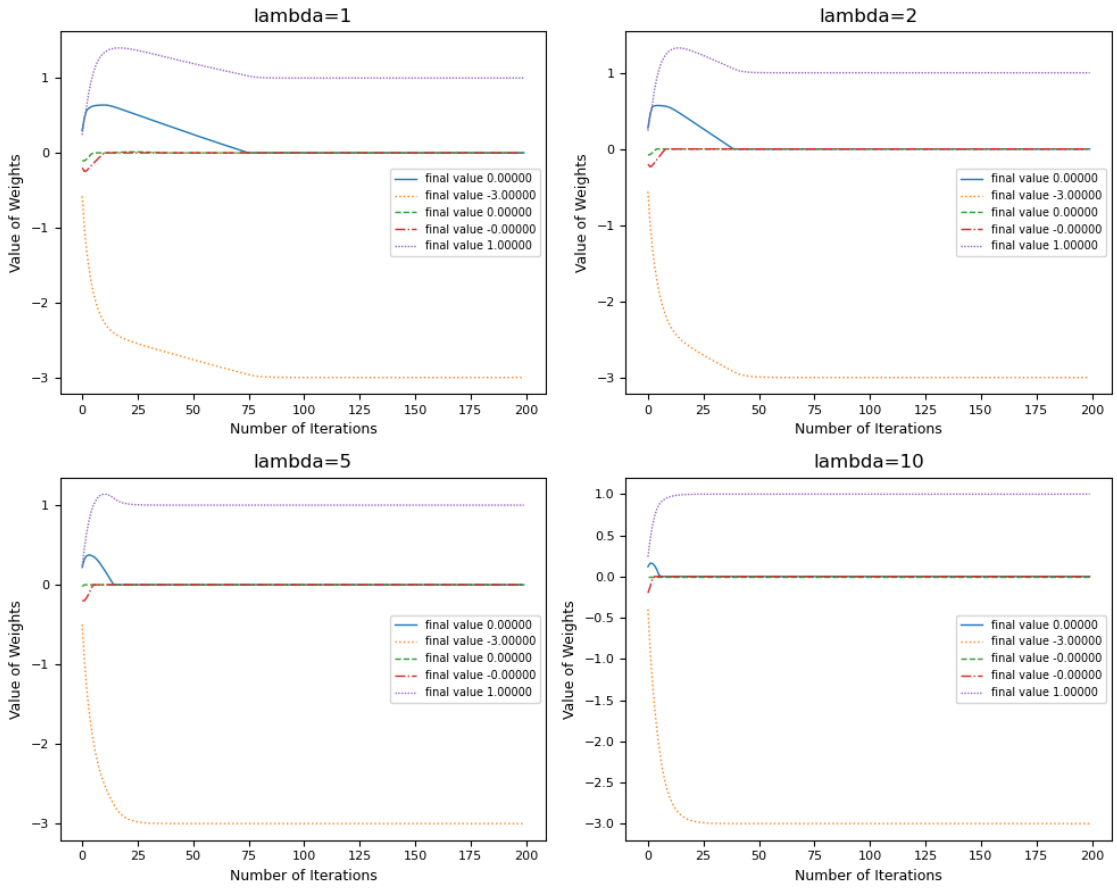
```
                                        l1_reg_params=lambda_list,␣
↪save_name_='lab1e_loss', save=True)
```



[309]:
```
"""(f) (6 pts) Trimmed l1 (Tl1) regulariser is proposed to solve the "bias"␣
↪problem of l1. For simplicity you may implement the T l1 regulariser as␣
↪applying a l1 regularization with strength  on the 3 elements of W with the␣
↪smallest absolute value, with no penalty on other elements. Minimize L +  T␣
↪l1 (W ) using proximal gradient update with  = {1.0, 2.0, 5.0, 10.0}␣
↪(correspond the soft thresholding threshold {0.02, 0.04, 0.1, 0.2}). Plot␣
↪the value of log(L) throughout the training, and use another figure to plot␣
↪the value of each element in W in each step. Comment on the convergence␣
↪comparison of the Trimmed l1 and the l1. Also compare the behavior of the␣
↪early steps (e.g. first 20) between the Trimmed l1 and the iterative pruning.
↪"""
lambda_list_f = np.array([1.0, 2.0, 5.0, 10.0]) * 1

weight_trim_list, loss_trim_list, weight_arr_trim_list = [], [], []
for idx, lll in enumerate(lambda_list_f):
```

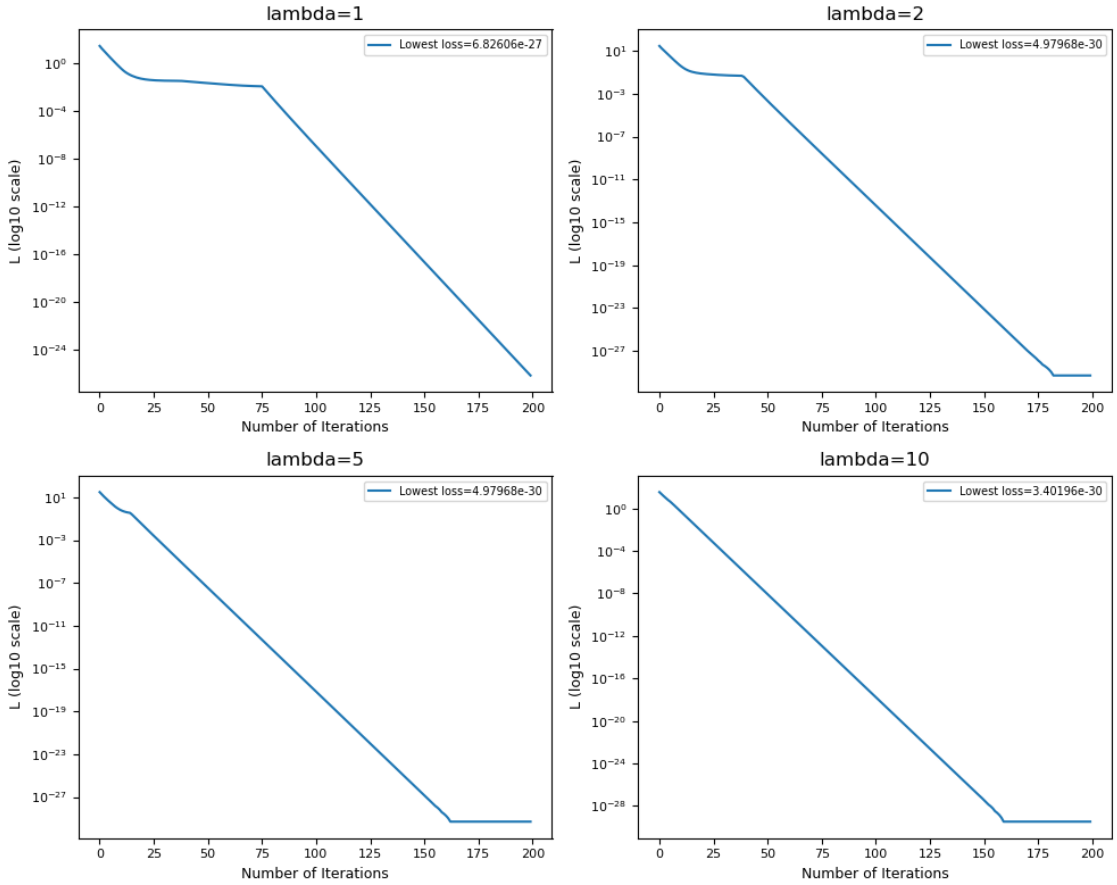
```
    w, l, w_array = run_grad_descent(w0, mu, x_input, y_input, n_epochs,␣
↪w0_prune=False, thresh=2,
                                     l1_reg_param=lll,␣
↪grad_fn=l1_norm_trimmed_update)
    weight_trim_list.append(w)
    loss_trim_list.append(l)
    weight_arr_trim_list.append(w_array)
```

[310]:
```
plot_many_w_vals_or_loss_vs_n_iters(num_epochs=n_epochs, w_init=w0,␣
↪final_weights_list=weight_trim_list,
                                    weight_array_list=weight_arr_trim_list,␣
↪loss_arr_lst=None,
                                    l1_reg_params=lambda_list_f,␣
↪save_name_='lab1f_weight', save=True)
```



[311]:
```
plot_many_w_vals_or_loss_vs_n_iters(num_epochs=n_epochs, w_init=w0,␣
↪final_weights_list=None,
                                    weight_array_list=None,␣
↪loss_arr_lst=loss_trim_list,
```

```
                                    l1_reg_params=lambda_list_f,␣
↪save_name_='lab1f_loss', save=True)
```



hw4

November 8, 2023

0.0.1 Lab2 (a) Model preperation

[21]:
```
from resnet20 import ResNetCIFAR
from train_util import train, finetune, test
import torch
import numpy as np
import matplotlib.pyplot as plt

import time

import torchvision.transforms as transforms
import torchvision
import torch.nn as nn
import torch.optim as optim

from FP_layers import *

device = 'cuda' if torch.cuda.is_available() else 'cpu'
device
```

[21]:
```
'cuda'
```

[22]:
```
net = ResNetCIFAR(num_layers=20, Nbits=None)
net = net.to(device)
```

[3]:
```
# Load the best weight paramters
net.load_state_dict(torch.load("pretrained_model.pt"))
test(net)
```

```
Files already downloaded and verified
Test Loss=0.3231, Test accuracy=0.9150
```

[3]:
```
0.915
```

0.0.2 Lab2 (b) Prune by percentage

[4]:
```
def prune_by_percentage(layer, q):
    """
    Pruning the weight paramters by threshold.
    :param q: pruning percentile. 'q' percent of the least
    significant weight parameters will be pruned.
    """
    with torch.no_grad():
        # Convert the weight of "layer" to numpy array
        layer_weight = layer.weight.detach().cpu().numpy()
        # Compute the q-th percentile of the abs of the converted array
        percentile = np.percentile(np.abs(layer_weight.flatten()), q)
        # Generate a binary mask same shape as weight to decide which element␣
↪to prune
        masked_obj = np.ma.masked_greater_equal(x=np.abs(layer_weight),␣
↪value=percentile, copy=True)
        mask_int = np.ma.getmask(masked_obj).astype(int)
        # Convert mask to torch tensor and put on GPU
        mask_tensor = torch.tensor(mask_int).to(device)
        # Multiply the weight by mask to perform pruning
        assert mask_int.shape == layer_weight.shape
        # layer.weight.data = mask_tensor * layer.weight.data
        layer.weight.data = layer.weight.data.clone().detach().
↪requires_grad_(True) * mask_tensor
    return
```

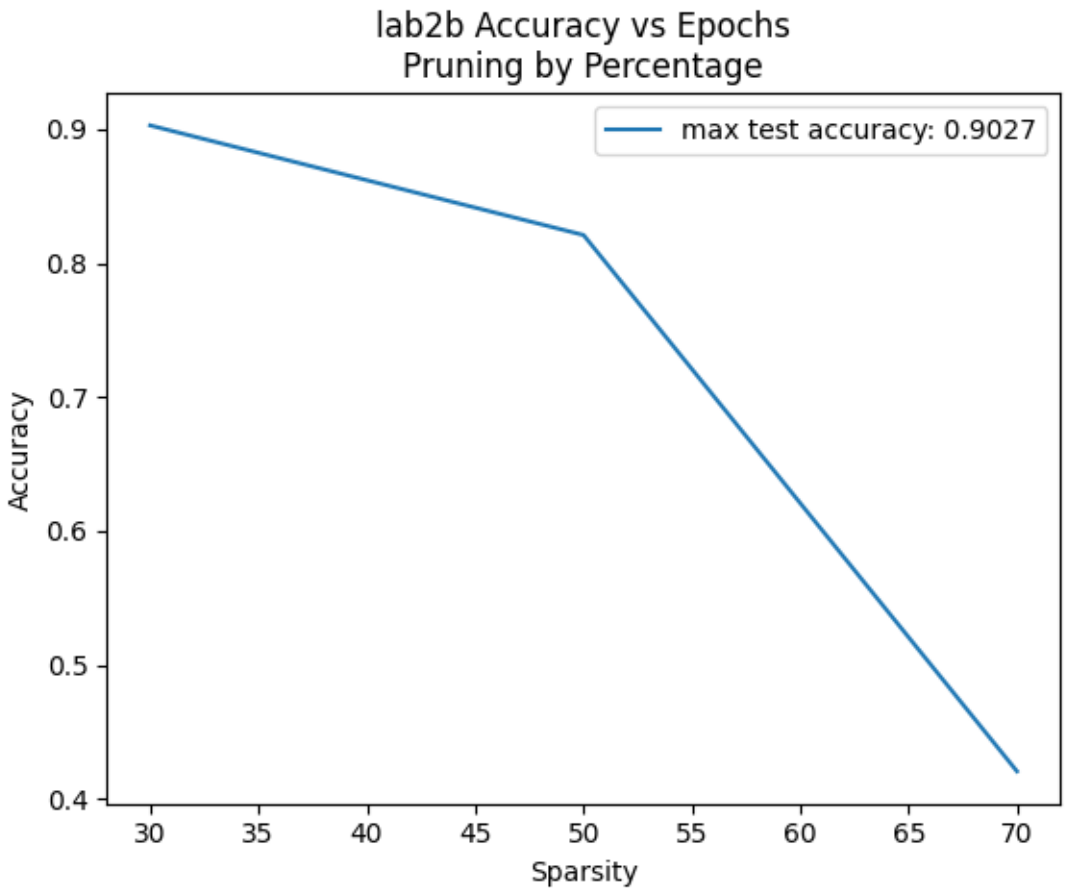
[5]:
```
q_list = np.array([0.3, 0.5, 0.7]) * 100

test_acc_prune_by_perc = []

for q_val in q_list:
    net.load_state_dict(torch.load("pretrained_model.pt"))
    for name, layer in net.named_modules():
        if (isinstance(layer, nn.Conv2d) or isinstance(layer, nn.Linear)) and␣
↪'id_mapping' not in name:
            # change q value
            prune_by_percentage(layer, q=q_val)
            # break
            ## Optional: Check the sparsity you achieve in each layer
            ## Convert the weight of "layer" to numpy array
            np_weight = layer.weight.detach().cpu().numpy()
            ## Count number of zeros
            zeros = sum((np_weight == 0).flatten())
            ## Count number of parameters
            total = len(np_weight.flatten())
```

```
            ## Print sparsity
            # print('Sparsity of ' + name + ': '+ str(zeros/total))
            print('Sparsity of %s: %g' % (name, zeros/total))
    test_acc_prune_by_perc.append(test(net))
```

```
Sparsity of head_conv.0.conv: 0.300926
Sparsity of body_op.0.conv1.0.conv: 0.299913
Sparsity of body_op.0.conv2.0.conv: 0.299913
Sparsity of body_op.1.conv1.0.conv: 0.299913
Sparsity of body_op.1.conv2.0.conv: 0.299913
Sparsity of body_op.2.conv1.0.conv: 0.299913
Sparsity of body_op.2.conv2.0.conv: 0.299913
Sparsity of body_op.3.conv1.0.conv: 0.30013
Sparsity of body_op.3.conv2.0.conv: 0.300022
Sparsity of body_op.4.conv1.0.conv: 0.300022
Sparsity of body_op.4.conv2.0.conv: 0.300022
Sparsity of body_op.5.conv1.0.conv: 0.300022
Sparsity of body_op.5.conv2.0.conv: 0.300022
Sparsity of body_op.6.conv1.0.conv: 0.300022
Sparsity of body_op.6.conv2.0.conv: 0.299995
Sparsity of body_op.7.conv1.0.conv: 0.299995
Sparsity of body_op.7.conv2.0.conv: 0.299995
Sparsity of body_op.8.conv1.0.conv: 0.299995
Sparsity of body_op.8.conv2.0.conv: 0.299995
Sparsity of final_fc.linear: 0.3
Files already downloaded and verified
Test Loss=0.3699, Test accuracy=0.9027
Sparsity of head_conv.0.conv: 0.5
Sparsity of body_op.0.conv1.0.conv: 0.5
Sparsity of body_op.0.conv2.0.conv: 0.5
Sparsity of body_op.1.conv1.0.conv: 0.5
Sparsity of body_op.1.conv2.0.conv: 0.5
Sparsity of body_op.2.conv1.0.conv: 0.5
Sparsity of body_op.2.conv2.0.conv: 0.5
Sparsity of body_op.3.conv1.0.conv: 0.5
Sparsity of body_op.3.conv2.0.conv: 0.5
Sparsity of body_op.4.conv1.0.conv: 0.5
Sparsity of body_op.4.conv2.0.conv: 0.5
Sparsity of body_op.5.conv1.0.conv: 0.5
Sparsity of body_op.5.conv2.0.conv: 0.5
Sparsity of body_op.6.conv1.0.conv: 0.5
Sparsity of body_op.6.conv2.0.conv: 0.5
Sparsity of body_op.7.conv1.0.conv: 0.5
Sparsity of body_op.7.conv2.0.conv: 0.5
Sparsity of body_op.8.conv1.0.conv: 0.5
Sparsity of body_op.8.conv2.0.conv: 0.5
Sparsity of final_fc.linear: 0.5
```

```
Files already downloaded and verified
Test Loss=0.6776, Test accuracy=0.8209
Sparsity of head_conv.0.conv: 0.699074
Sparsity of body_op.0.conv1.0.conv: 0.700087
Sparsity of body_op.0.conv2.0.conv: 0.700087
Sparsity of body_op.1.conv1.0.conv: 0.700087
Sparsity of body_op.1.conv2.0.conv: 0.700087
Sparsity of body_op.2.conv1.0.conv: 0.700087
Sparsity of body_op.2.conv2.0.conv: 0.700087
Sparsity of body_op.3.conv1.0.conv: 0.69987
Sparsity of body_op.3.conv2.0.conv: 0.699978
Sparsity of body_op.4.conv1.0.conv: 0.699978
Sparsity of body_op.4.conv2.0.conv: 0.699978
Sparsity of body_op.5.conv1.0.conv: 0.699978
Sparsity of body_op.5.conv2.0.conv: 0.699978
Sparsity of body_op.6.conv1.0.conv: 0.699978
Sparsity of body_op.6.conv2.0.conv: 0.700005
Sparsity of body_op.7.conv1.0.conv: 0.700005
Sparsity of body_op.7.conv2.0.conv: 0.700005
Sparsity of body_op.8.conv1.0.conv: 0.700005
Sparsity of body_op.8.conv2.0.conv: 0.700005
Sparsity of final_fc.linear: 0.7
Files already downloaded and verified
Test Loss=2.4418, Test accuracy=0.4207
```

[6]:
```
fig, ax = plt.subplots(1, 1)
ax.plot(q_list, test_acc_prune_by_perc, label='max test accuracy: %g' % np.
  ↪max(test_acc_prune_by_perc))
ax.set_xlabel('Sparsity')
ax.set_ylabel('Accuracy')
ax.set_title('lab2b Accuracy vs Epochs\nPruning by Percentage')
ax.legend()

plt.savefig('lab2b.pdf', dpi=500, bbox_inches='tight')
```



0.0.3 Lab2 (c) Finetune pruned model

[7]:
```
def finetune_after_prune(net, trainloader, criterion, optimizer, prune=True):
    """
    Finetune the pruned model for a single epoch
    Make sure pruned weights are kept as zero
    """
    # Build a dictionary for the nonzero weights
    weight_mask = {}
    for name, layer in net.named_modules():
        if (isinstance(layer, nn.Conv2d) or isinstance(layer, nn.Linear)) and␣
↪'id_mapping' not in name:
            # Your code here: generate a mask in GPU torch tensor to have 1 for␣
↪nonzero element and 0 for zero element
            layer_weight = layer.weight.detach().cpu().numpy()
            mask_obj = np.ma.masked_equal(x=layer_weight, value=0)  # where␣
↪it's zero, mask value is True
```

```
            weight_mask[name] = torch.tensor(1 - np.ma.getmask(mask_obj).
↪astype(int)).to(device)
            # zero = 0, nonzero = 1

    global_steps = 0
    train_loss = 0
    correct = 0
    total = 0
    start = time.time()
    for batch_idx, (inputs, targets) in enumerate(trainloader):
        inputs, targets = inputs.to(device), targets.to(device)
        optimizer.zero_grad()
        outputs = net(inputs)
        loss = criterion(outputs, targets)
        loss.backward()
        optimizer.step()

        if prune:
            for name,layer in net.named_modules():
                if (isinstance(layer, nn.Conv2d) or isinstance(layer, nn.
↪Linear)) and 'id_mapping' not in name:
                    # Your code here: Use weight_mask to make sure zero␣
↪elements remains zero
                    layer.weight.data = layer.weight.data.clone().detach().
↪requires_grad_(True) * weight_mask[name]

        train_loss += loss.item()
        _, predicted = outputs.max(1)
        total += targets.size(0)
        correct += predicted.eq(targets).sum().item()
        global_steps += 1

        if global_steps % 50 == 0:
            end = time.time()
            batch_size = 256
            num_examples_per_second = 50 * batch_size / (end - start)
            print("[Step=%d]\tLoss=%.4f\tacc=%.4f\t%.1f examples/second"
                  % (global_steps, train_loss / (batch_idx + 1), (correct /␣
↪total), num_examples_per_second))
            start = time.time()
```

[8]:
```
# Get pruned model
net.load_state_dict(torch.load("pretrained_model.pt"))
for name, layer in net.named_modules():
    if (isinstance(layer, nn.Conv2d) or isinstance(layer, nn.Linear)) and␣
↪'id_mapping' not in name:
```

```
        prune_by_percentage(layer, q=70.0)
lab2c_test_b4_finetune = test(net)
# Training setup, do not change
batch_size = 256
lr = 0.002
reg = 1e-4

print('==> Preparing data..')
transform_train = transforms.Compose([
    transforms.RandomCrop(32, padding=4),
    transforms.RandomHorizontalFlip(),
    transforms.ToTensor(),
    transforms.Normalize((0.4914, 0.4822, 0.4465), (0.2023, 0.1994, 0.2010)),
])

transform_test = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.4914, 0.4822, 0.4465), (0.2023, 0.1994, 0.2010)),
])
best_acc = 0  # best test accuracy
start_epoch = 0  # start from epoch 0 or last checkpoint epoch
trainset = torchvision.datasets.CIFAR10(root='./data', train=True,␣
 ↪download=True, transform=transform_train)
trainloader = torch.utils.data.DataLoader(trainset, batch_size=batch_size,␣
 ↪shuffle=True, num_workers=16)

testset = torchvision.datasets.CIFAR10(root='./data', train=False,␣
 ↪download=True, transform=transform_test)
testloader = torch.utils.data.DataLoader(testset, batch_size=100,␣
 ↪shuffle=False, num_workers=2)

criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(net.parameters(), lr=lr, momentum=0.875,␣
 ↪weight_decay=reg, nesterov=False)
```

```
Files already downloaded and verified
Test Loss=2.4418, Test accuracy=0.4207
==> Preparing data..
Files already downloaded and verified
Files already downloaded and verified
```

[9]:
```
# Model finetuning
test_acc_lst = []
n_epochs = 20
for epoch in range(n_epochs):
    print('\nEpoch: %d' % epoch)
    net.train()
```

```
    finetune_after_prune(net, trainloader, criterion, optimizer)
    #Start the testing code.
    net.eval()
    test_loss = 0
    correct = 0
    total = 0
    with torch.no_grad():
        for batch_idx, (inputs, targets) in enumerate(testloader):
            inputs, targets = inputs.to(device), targets.to(device)
            outputs = net(inputs)
            loss = criterion(outputs, targets)

            test_loss += loss.item()
            _, predicted = outputs.max(1)
            total += targets.size(0)
            correct += predicted.eq(targets).sum().item()
    num_val_steps = len(testloader)
    val_acc = correct / total
    print("Test Loss=%.4f, Test acc=%.4f" % (test_loss / (num_val_steps),␣
↪val_acc))
    test_acc_lst.append(val_acc)
    if val_acc > best_acc:
        best_acc = val_acc
        print("Saving...")
        torch.save(net.state_dict(), "net_after_finetune.pt")
```

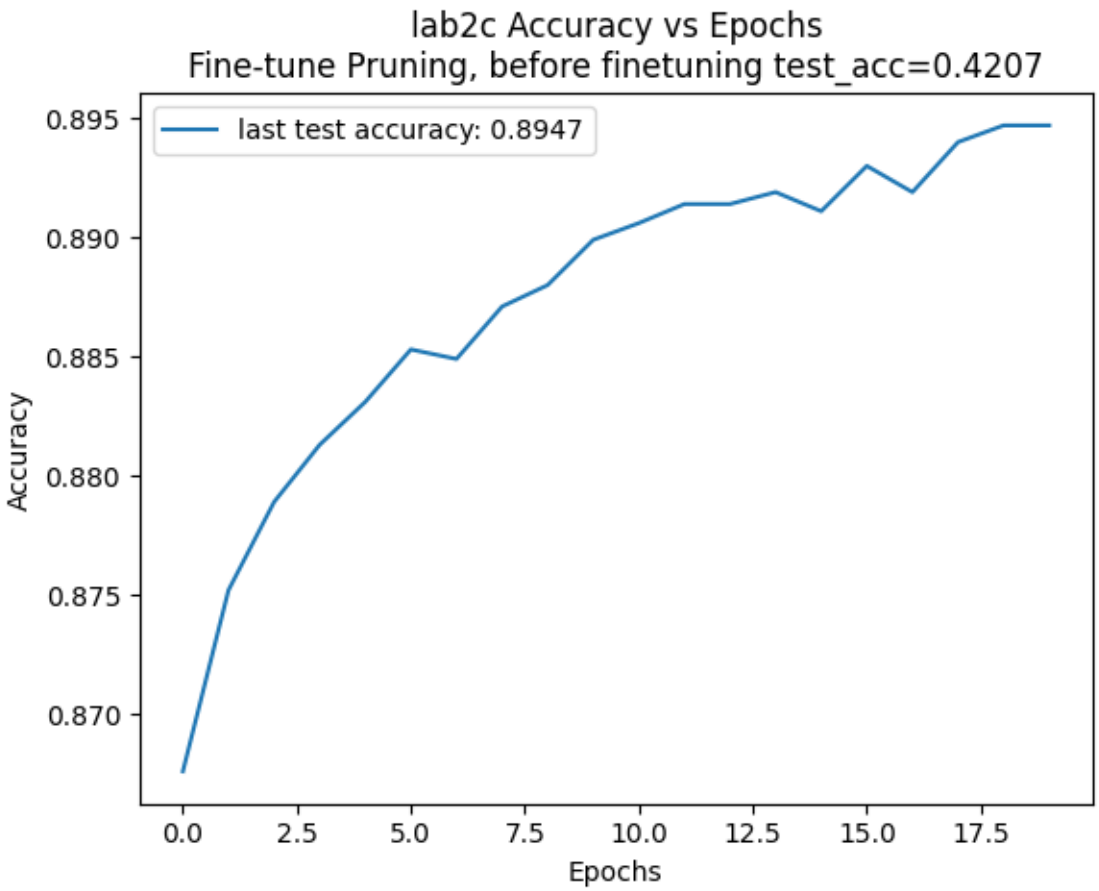
```
Epoch: 0
[Step=50]       Loss=0.4015     acc=0.8612      6063.1 examples/second
[Step=100]      Loss=0.3621     acc=0.8752      12533.4 examples/second
[Step=150]      Loss=0.3334     acc=0.8853      12582.2 examples/second
Test Loss=0.4290, Test acc=0.8676
Saving…

Epoch: 1
[Step=50]       Loss=0.2458     acc=0.9155      6556.5 examples/second
[Step=100]      Loss=0.2449     acc=0.9152      12105.4 examples/second
[Step=150]      Loss=0.2388     acc=0.9171      12214.2 examples/second
Test Loss=0.3941, Test acc=0.8752
Saving…

Epoch: 2
[Step=50]       Loss=0.2220     acc=0.9273      6774.9 examples/second
[Step=100]      Loss=0.2218     acc=0.9254      12616.7 examples/second
[Step=150]      Loss=0.2150     acc=0.9276      12099.2 examples/second
Test Loss=0.3787, Test acc=0.8789
Saving…
```

```
Epoch: 3
[Step=50]       Loss=0.1934     acc=0.9318      6663.5 examples/second
[Step=100]      Loss=0.2036     acc=0.9284      12776.3 examples/second
[Step=150]      Loss=0.2007     acc=0.9292      12756.8 examples/second
Test Loss=0.3710, Test acc=0.8813
Saving…

Epoch: 4
[Step=50]       Loss=0.1908     acc=0.9345      6193.6 examples/second
[Step=100]      Loss=0.1856     acc=0.9364      13214.0 examples/second
[Step=150]      Loss=0.1860     acc=0.9357      13242.3 examples/second
Test Loss=0.3622, Test acc=0.8831
Saving…

Epoch: 5
[Step=50]       Loss=0.1849     acc=0.9354      6596.6 examples/second
[Step=100]      Loss=0.1848     acc=0.9354      12605.2 examples/second
[Step=150]      Loss=0.1826     acc=0.9359      12794.9 examples/second
Test Loss=0.3581, Test acc=0.8853
Saving…

Epoch: 6
[Step=50]       Loss=0.1807     acc=0.9342      6581.0 examples/second
[Step=100]      Loss=0.1772     acc=0.9370      12638.2 examples/second
[Step=150]      Loss=0.1771     acc=0.9373      12780.2 examples/second
Test Loss=0.3544, Test acc=0.8849

Epoch: 7
[Step=50]       Loss=0.1696     acc=0.9415      6120.6 examples/second
[Step=100]      Loss=0.1741     acc=0.9398      11072.2 examples/second
[Step=150]      Loss=0.1734     acc=0.9403      12073.7 examples/second
Test Loss=0.3505, Test acc=0.8871
Saving…

Epoch: 8
[Step=50]       Loss=0.1747     acc=0.9388      5946.8 examples/second
[Step=100]      Loss=0.1668     acc=0.9422      11545.4 examples/second
[Step=150]      Loss=0.1655     acc=0.9422      12119.1 examples/second
Test Loss=0.3488, Test acc=0.8880
Saving…

Epoch: 9
[Step=50]       Loss=0.1605     acc=0.9440      6828.1 examples/second
[Step=100]      Loss=0.1617     acc=0.9445      13478.8 examples/second
[Step=150]      Loss=0.1620     acc=0.9444      13561.5 examples/second
Test Loss=0.3445, Test acc=0.8899
Saving…
```

```
Epoch: 10
[Step=50]       Loss=0.1613     acc=0.9432      6316.5 examples/second
[Step=100]      Loss=0.1633     acc=0.9413      12868.6 examples/second
[Step=150]      Loss=0.1629     acc=0.9414      13039.7 examples/second
Test Loss=0.3437, Test acc=0.8906
Saving…

Epoch: 11
[Step=50]       Loss=0.1552     acc=0.9457      6349.5 examples/second
[Step=100]      Loss=0.1576     acc=0.9442      12022.5 examples/second
[Step=150]      Loss=0.1589     acc=0.9437      13227.5 examples/second
Test Loss=0.3418, Test acc=0.8914
Saving…

Epoch: 12
[Step=50]       Loss=0.1499     acc=0.9504      6148.7 examples/second
[Step=100]      Loss=0.1527     acc=0.9482      12449.1 examples/second
[Step=150]      Loss=0.1522     acc=0.9489      12410.9 examples/second
Test Loss=0.3398, Test acc=0.8914

Epoch: 13
[Step=50]       Loss=0.1548     acc=0.9458      6635.9 examples/second
[Step=100]      Loss=0.1583     acc=0.9450      12242.1 examples/second
[Step=150]      Loss=0.1555     acc=0.9462      12662.9 examples/second
Test Loss=0.3404, Test acc=0.8919
Saving…

Epoch: 14
[Step=50]       Loss=0.1570     acc=0.9452      6721.6 examples/second
[Step=100]      Loss=0.1480     acc=0.9491      12571.6 examples/second
[Step=150]      Loss=0.1514     acc=0.9472      12635.6 examples/second
Test Loss=0.3387, Test acc=0.8911

Epoch: 15
[Step=50]       Loss=0.1461     acc=0.9513      6599.7 examples/second
[Step=100]      Loss=0.1475     acc=0.9502      12253.1 examples/second
[Step=150]      Loss=0.1499     acc=0.9492      12467.3 examples/second
Test Loss=0.3371, Test acc=0.8930
Saving…

Epoch: 16
[Step=50]       Loss=0.1482     acc=0.9480      5790.8 examples/second
[Step=100]      Loss=0.1453     acc=0.9493      12149.8 examples/second
[Step=150]      Loss=0.1480     acc=0.9482      12349.3 examples/second
Test Loss=0.3375, Test acc=0.8919

Epoch: 17
```

```
[Step=50]       Loss=0.1414     acc=0.9503      5905.5 examples/second
[Step=100]      Loss=0.1441     acc=0.9495      12511.9 examples/second
[Step=150]      Loss=0.1428     acc=0.9504      12075.0 examples/second
Test Loss=0.3341, Test acc=0.8940
Saving…

Epoch: 18
[Step=50]       Loss=0.1442     acc=0.9498      6657.1 examples/second
[Step=100]      Loss=0.1476     acc=0.9484      12269.5 examples/second
[Step=150]      Loss=0.1461     acc=0.9492      12533.2 examples/second
Test Loss=0.3343, Test acc=0.8947
Saving…

Epoch: 19
[Step=50]       Loss=0.1399     acc=0.9519      6243.7 examples/second
[Step=100]      Loss=0.1443     acc=0.9498      12571.7 examples/second
[Step=150]      Loss=0.1426     acc=0.9509      12707.4 examples/second
Test Loss=0.3344, Test acc=0.8947
```

[10]:
```
fig, ax = plt.subplots(1, 1)
xx = range(n_epochs)
ax.plot(xx, test_acc_lst, label='last test accuracy: %g' % test_acc_lst[-1])
ax.set_xlabel('Epochs')
ax.set_ylabel('Accuracy')
ax.set_title('lab2c Accuracy vs Epochs\nFine-tune Pruning, before finetuning␣
 ↪test_acc=%.4f'
             % lab2c_test_b4_finetune)
ax.legend()

plt.savefig('Figures/lab2c.pdf', dpi=500, bbox_inches='tight')
```



[11]:
```
# Check sparsity of the finetuned model, make sure it's not changed
net.load_state_dict(torch.load("net_after_finetune.pt"))

for name,layer in net.named_modules():
    if (isinstance(layer, nn.Conv2d) or isinstance(layer, nn.Linear)) and␣
 ↪'id_mapping' not in name:
        # Your code here:
        ## Convert the weight of "layer" to numpy array
        np_weight = layer.weight.detach().cpu().numpy()
        ## Count number of zeros
        zeros = sum((np_weight == 0).flatten())
        ## Count number of parameters
        total = len(np_weight.flatten())
        # Print sparsity
        print('Sparsity of %s: %g' % (name, zeros/total))

test(net)
```

```
Sparsity of head_conv.0.conv: 0.699074
```

```
Sparsity of body_op.0.conv1.0.conv: 0.700087
Sparsity of body_op.0.conv2.0.conv: 0.700087
Sparsity of body_op.1.conv1.0.conv: 0.700087
Sparsity of body_op.1.conv2.0.conv: 0.700087
Sparsity of body_op.2.conv1.0.conv: 0.700087
Sparsity of body_op.2.conv2.0.conv: 0.700087
Sparsity of body_op.3.conv1.0.conv: 0.69987
Sparsity of body_op.3.conv2.0.conv: 0.699978
Sparsity of body_op.4.conv1.0.conv: 0.699978
Sparsity of body_op.4.conv2.0.conv: 0.699978
Sparsity of body_op.5.conv1.0.conv: 0.699978
Sparsity of body_op.5.conv2.0.conv: 0.699978
Sparsity of body_op.6.conv1.0.conv: 0.699978
Sparsity of body_op.6.conv2.0.conv: 0.700005
Sparsity of body_op.7.conv1.0.conv: 0.700005
Sparsity of body_op.7.conv2.0.conv: 0.700005
Sparsity of body_op.8.conv1.0.conv: 0.700005
Sparsity of body_op.8.conv2.0.conv: 0.700005
Sparsity of final_fc.linear: 0.7
Files already downloaded and verified
Test Loss=0.3343, Test accuracy=0.8947
```

[11]:
```
0.8947
```

0.0.4 Lab2 (d) Iterative pruning

[12]:
```
net.load_state_dict(torch.load("pretrained_model.pt"))
best_acc = 0.
n_epochs = 20

test_acc_iter_prune = []

for epoch in range(n_epochs):
    print('\nEpoch: %d' % epoch)

    net.train()
    if epoch < 10:
        for name,layer in net.named_modules():
            if (isinstance(layer, nn.Conv2d) or isinstance(layer, nn.Linear))␣
 ↪and 'id_mapping' not in name:
                # Increase model sparsity
                q = (epoch + 1) * 7
                # Linearly increase the pruning percentage for 10 epochs until␣
 ↪reaching 70% in the final epoch
                prune_by_percentage(layer, q=q)
    if epoch < 9:
```

```
        finetune_after_prune(net, trainloader, criterion, optimizer,␣
↪prune=False)
    else: # starts from epoch==9, where q = 70%
        finetune_after_prune(net, trainloader, criterion, optimizer)

    #Start the testing code.
    net.eval()
    test_loss = 0
    correct = 0
    total = 0
    with torch.no_grad():
        for batch_idx, (inputs, targets) in enumerate(testloader):
            inputs, targets = inputs.to(device), targets.to(device)
            outputs = net(inputs)
            loss = criterion(outputs, targets)

            test_loss += loss.item()
            _, predicted = outputs.max(1)
            total += targets.size(0)
            correct += predicted.eq(targets).sum().item()
    num_val_steps = len(testloader)
    val_acc = correct / total

    test_acc_iter_prune.append(val_acc)

    print("Test Loss=%.4f, Test acc=%.4f" % (test_loss / (num_val_steps),␣
↪val_acc))

    if epoch>=10:
        if val_acc > best_acc:
            best_acc = val_acc
            print("Saving...")
            torch.save(net.state_dict(), "net_after_iterative_prune.pt")
```

```
Epoch: 0
[Step=50]       Loss=0.0472     acc=0.9847      6590.0 examples/second
[Step=100]      Loss=0.0482     acc=0.9841      12078.6 examples/second
[Step=150]      Loss=0.0481     acc=0.9843      12069.6 examples/second
Test Loss=0.3262, Test acc=0.9134

Epoch: 1
[Step=50]       Loss=0.0482     acc=0.9840      6651.5 examples/second
[Step=100]      Loss=0.0497     acc=0.9833      12706.2 examples/second
[Step=150]      Loss=0.0481     acc=0.9841      12666.0 examples/second
Test Loss=0.3265, Test acc=0.9139
```

```
Epoch: 2
[Step=50]       Loss=0.0475     acc=0.9846      6783.0 examples/second
[Step=100]      Loss=0.0507     acc=0.9833      12782.9 examples/second
[Step=150]      Loss=0.0497     acc=0.9841      12842.8 examples/second
Test Loss=0.3286, Test acc=0.9148

Epoch: 3
[Step=50]       Loss=0.0521     acc=0.9824      5565.0 examples/second
[Step=100]      Loss=0.0516     acc=0.9832      11161.7 examples/second
[Step=150]      Loss=0.0525     acc=0.9824      12087.6 examples/second
Test Loss=0.3304, Test acc=0.9125

Epoch: 4
[Step=50]       Loss=0.0631     acc=0.9784      6435.7 examples/second
[Step=100]      Loss=0.0618     acc=0.9793      12608.6 examples/second
[Step=150]      Loss=0.0608     acc=0.9797      12673.7 examples/second
Test Loss=0.3343, Test acc=0.9117

Epoch: 5
[Step=50]       Loss=0.0740     acc=0.9750      6680.0 examples/second
[Step=100]      Loss=0.0719     acc=0.9757      12194.1 examples/second
[Step=150]      Loss=0.0700     acc=0.9764      12545.0 examples/second
Test Loss=0.3380, Test acc=0.9084

Epoch: 6
[Step=50]       Loss=0.0918     acc=0.9689      6544.7 examples/second
[Step=100]      Loss=0.0897     acc=0.9691      12636.3 examples/second
[Step=150]      Loss=0.0878     acc=0.9701      12305.3 examples/second
Test Loss=0.3330, Test acc=0.9063

Epoch: 7
[Step=50]       Loss=0.1278     acc=0.9573      6697.8 examples/second
[Step=100]      Loss=0.1239     acc=0.9588      12323.9 examples/second
[Step=150]      Loss=0.1204     acc=0.9596      12597.7 examples/second
Test Loss=0.3359, Test acc=0.9022

Epoch: 8
[Step=50]       Loss=0.1709     acc=0.9405      6641.5 examples/second
[Step=100]      Loss=0.1636     acc=0.9422      12175.7 examples/second
[Step=150]      Loss=0.1570     acc=0.9448      12565.0 examples/second
Test Loss=0.3472, Test acc=0.8941

Epoch: 9
[Step=50]       Loss=0.2701     acc=0.9071      6561.1 examples/second
[Step=100]      Loss=0.2503     acc=0.9139      12533.8 examples/second
[Step=150]      Loss=0.2406     acc=0.9166      12643.9 examples/second
Test Loss=0.3901, Test acc=0.8759
```

```
Epoch: 10
[Step=50]       Loss=0.2112     acc=0.9237      6791.6 examples/second
[Step=100]      Loss=0.2032     acc=0.9277      12589.8 examples/second
[Step=150]      Loss=0.2042     acc=0.9279      12803.1 examples/second
Test Loss=0.3735, Test acc=0.8804
Saving…

Epoch: 11
[Step=50]       Loss=0.1868     acc=0.9356      6858.2 examples/second
[Step=100]      Loss=0.1856     acc=0.9359      12521.4 examples/second
[Step=150]      Loss=0.1862     acc=0.9343      11670.0 examples/second
Test Loss=0.3674, Test acc=0.8825
Saving…

Epoch: 12
[Step=50]       Loss=0.1798     acc=0.9384      6543.2 examples/second
[Step=100]      Loss=0.1781     acc=0.9386      12376.5 examples/second
[Step=150]      Loss=0.1805     acc=0.9376      13172.6 examples/second
Test Loss=0.3585, Test acc=0.8853
Saving…

Epoch: 13
[Step=50]       Loss=0.1798     acc=0.9373      5819.9 examples/second
[Step=100]      Loss=0.1740     acc=0.9392      10813.6 examples/second
[Step=150]      Loss=0.1709     acc=0.9397      10752.0 examples/second
Test Loss=0.3546, Test acc=0.8866
Saving…

Epoch: 14
[Step=50]       Loss=0.1670     acc=0.9427      6128.3 examples/second
[Step=100]      Loss=0.1696     acc=0.9418      11742.8 examples/second
[Step=150]      Loss=0.1661     acc=0.9427      12339.2 examples/second
Test Loss=0.3511, Test acc=0.8874
Saving…

Epoch: 15
[Step=50]       Loss=0.1665     acc=0.9420      6126.5 examples/second
[Step=100]      Loss=0.1631     acc=0.9429      12471.9 examples/second
[Step=150]      Loss=0.1645     acc=0.9425      12356.1 examples/second
Test Loss=0.3488, Test acc=0.8893
Saving…

Epoch: 16
[Step=50]       Loss=0.1545     acc=0.9454      5152.9 examples/second
[Step=100]      Loss=0.1574     acc=0.9453      11185.4 examples/second
[Step=150]      Loss=0.1581     acc=0.9445      11841.8 examples/second
Test Loss=0.3450, Test acc=0.8899
Saving…
```

```
Epoch: 17
[Step=50]       Loss=0.1604     acc=0.9439      6179.6 examples/second
[Step=100]      Loss=0.1543     acc=0.9450      11965.1 examples/second
[Step=150]      Loss=0.1543     acc=0.9456      11827.1 examples/second
Test Loss=0.3437, Test acc=0.8911
Saving…

Epoch: 18
[Step=50]       Loss=0.1557     acc=0.9444      6268.0 examples/second
[Step=100]      Loss=0.1548     acc=0.9464      12063.5 examples/second
[Step=150]      Loss=0.1541     acc=0.9463      12084.5 examples/second
Test Loss=0.3431, Test acc=0.8905

Epoch: 19
[Step=50]       Loss=0.1500     acc=0.9492      6166.7 examples/second
[Step=100]      Loss=0.1519     acc=0.9481      12370.6 examples/second
[Step=150]      Loss=0.1496     acc=0.9488      12396.4 examples/second
Test Loss=0.3405, Test acc=0.8922
Saving…
```

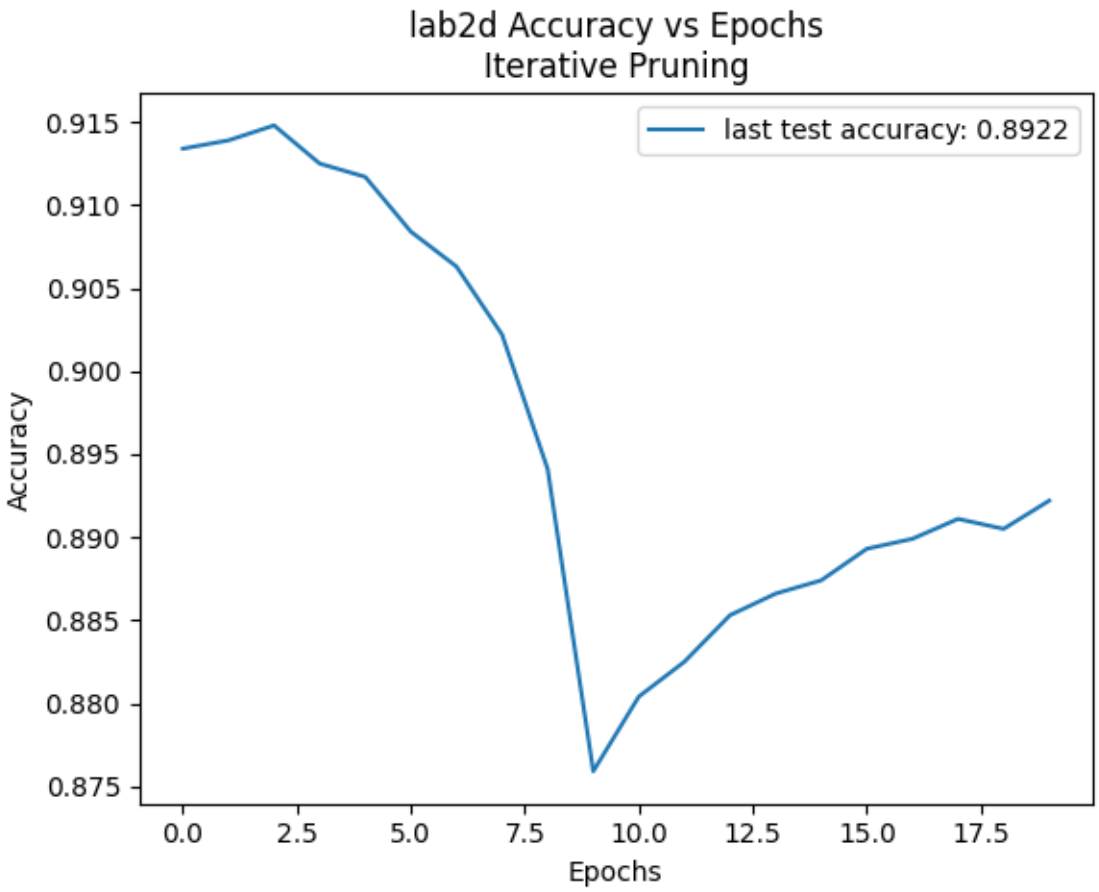
[13]:
```python
# Check sparsity of the final model, make sure it's 70%
net.load_state_dict(torch.load("net_after_iterative_prune.pt"))

for name,layer in net.named_modules():
    if (isinstance(layer, nn.Conv2d) or isinstance(layer, nn.Linear)) and␣
 ↪'id_mapping' not in name:
        # Your code here: can copy from previous question
        ## Convert the weight of "layer" to numpy array
        np_weight = layer.weight.detach().cpu().numpy()
        ## Count number of zeros
        zeros = sum((np_weight == 0).flatten())
        ## Count number of parameters
        total = len(np_weight.flatten())
        # Print sparsity
        print('Sparsity of %s: %g' % (name, zeros/total))


test(net)
```

```
Sparsity of head_conv.0.conv: 0.699074
Sparsity of body_op.0.conv1.0.conv: 0.700087
Sparsity of body_op.0.conv2.0.conv: 0.700087
Sparsity of body_op.1.conv1.0.conv: 0.700087
Sparsity of body_op.1.conv2.0.conv: 0.700087
Sparsity of body_op.2.conv1.0.conv: 0.700087
Sparsity of body_op.2.conv2.0.conv: 0.700087
Sparsity of body_op.3.conv1.0.conv: 0.69987
```

```
Sparsity of body_op.3.conv2.0.conv: 0.699978
Sparsity of body_op.4.conv1.0.conv: 0.699978
Sparsity of body_op.4.conv2.0.conv: 0.699978
Sparsity of body_op.5.conv1.0.conv: 0.699978
Sparsity of body_op.5.conv2.0.conv: 0.699978
Sparsity of body_op.6.conv1.0.conv: 0.699978
Sparsity of body_op.6.conv2.0.conv: 0.700005
Sparsity of body_op.7.conv1.0.conv: 0.700005
Sparsity of body_op.7.conv2.0.conv: 0.700005
Sparsity of body_op.8.conv1.0.conv: 0.700005
Sparsity of body_op.8.conv2.0.conv: 0.700005
Sparsity of final_fc.linear: 0.7
Files already downloaded and verified
Test Loss=0.3405, Test accuracy=0.8922
```

[13]:
```
0.8922
```

[14]:
```
fig, ax = plt.subplots(1, 1)
xx = range(n_epochs)
ax.plot(xx, test_acc_iter_prune, label='last test accuracy: %g' %␣
  ↪test_acc_iter_prune[-1])
ax.set_xlabel('Epochs')
ax.set_ylabel('Accuracy')
ax.set_title('lab2d Accuracy vs Epochs\nIterative Pruning')
ax.legend()

plt.savefig('Figures/lab2d.pdf', dpi=500, bbox_inches='tight')
```



0.0.5 Lab2 (e) Global iterative pruning

[15]:
```
import numpy as np

def global_prune_by_percentage(net, q=70.0):
    """
    Pruning the weight paramters by threshold.
    :param q: pruning percentile. 'q' percent of the least
    significant weight parameters will be pruned.
    """
    # A list to gather all the weights
    flattened_weights = []
    # Find global pruning threshold
    for name,layer in net.named_modules():
        if (isinstance(layer, nn.Conv2d) or isinstance(layer, nn.Linear)) and␣
↪'id_mapping' not in name:
            # Convert weight to numpy
            # Flatten the weight and append to flattened_weights
```

```
            flattened_weights.append(layer.weight.detach().cpu().numpy().
↪flatten())

    # Concate all weights into a np array
    flattened_weights = np.concatenate(flattened_weights)
    # Find global pruning threshold
    threshold = np.percentile(np.abs(flattened_weights.flatten()), q)

    # Apply pruning threshold to all layers
    for name,layer in net.named_modules():
        if (isinstance(layer, nn.Conv2d) or isinstance(layer, nn.Linear)) and␣
↪'id_mapping' not in name:
            # Convert weight to numpy
            layer_weight = layer.weight.detach().cpu().numpy()

            # Generate a binary mask same shape as weight to decide which␣
↪element to prune
            masked_obj = np.ma.masked_greater_equal(x=np.abs(layer_weight),␣
↪value=threshold, copy=True)
            mask_int = np.ma.getmask(masked_obj).astype(int)
            # Convert mask to torch tensor and put on GPU
            mask_tensor = torch.tensor(mask_int).to(device)
            # Multiply the weight by mask to perform pruning
            layer.weight.data = layer.weight.data.clone().detach().
↪requires_grad_(True) * mask_tensor
```

[16]:
```
net.load_state_dict(torch.load("pretrained_model.pt"))
best_acc = 0.

test_acc_global_prune = []

for epoch in range(20):
    print('\nEpoch: %d' % epoch)
    # q = (epoch + 1) * 8
    q = (epoch + 1) * 7

    net.train()
    # Increase model sparsity
    if epoch < 10:
        global_prune_by_percentage(net, q=q)
    if epoch < 9:
        finetune_after_prune(net, trainloader, criterion, optimizer,␣
↪prune=False)
    else:
        finetune_after_prune(net, trainloader, criterion, optimizer)

    #Start the testing code.
```

```
    net.eval()
    test_loss = 0
    correct = 0
    total = 0
    with torch.no_grad():
        for batch_idx, (inputs, targets) in enumerate(testloader):
            inputs, targets = inputs.to(device), targets.to(device)
            outputs = net(inputs)
            loss = criterion(outputs, targets)

            test_loss += loss.item()
            _, predicted = outputs.max(1)
            total += targets.size(0)
            correct += predicted.eq(targets).sum().item()
    num_val_steps = len(testloader)
    val_acc = correct / total
    print("Test Loss=%.4f, Test acc=%.4f" % (test_loss / (num_val_steps),␣
↪val_acc))

    test_acc_global_prune.append(val_acc)

    if epoch >= 10:
        if val_acc > best_acc:
            best_acc = val_acc
            print("Saving...")
            torch.save(net.state_dict(), "net_after_global_iterative_prune.pt")
```

```
Epoch: 0
[Step=50]       Loss=0.0494     acc=0.9846      6143.9 examples/second
[Step=100]      Loss=0.0484     acc=0.9847      12491.2 examples/second
[Step=150]      Loss=0.0493     acc=0.9843      12542.2 examples/second
Test Loss=0.3235, Test acc=0.9150

Epoch: 1
[Step=50]       Loss=0.0483     acc=0.9848      6045.3 examples/second
[Step=100]      Loss=0.0488     acc=0.9844      12603.3 examples/second
[Step=150]      Loss=0.0488     acc=0.9844      10154.3 examples/second
Test Loss=0.3239, Test acc=0.9143

Epoch: 2
[Step=50]       Loss=0.0480     acc=0.9859      5200.7 examples/second
[Step=100]      Loss=0.0480     acc=0.9857      12315.7 examples/second
[Step=150]      Loss=0.0483     acc=0.9854      12389.8 examples/second
Test Loss=0.3291, Test acc=0.9137

Epoch: 3
```

```
[Step=50]       Loss=0.0501     acc=0.9843      6317.4 examples/second
[Step=100]      Loss=0.0496     acc=0.9845      12250.7 examples/second
[Step=150]      Loss=0.0499     acc=0.9841      12418.7 examples/second
Test Loss=0.3274, Test acc=0.9127

Epoch: 4
[Step=50]       Loss=0.0500     acc=0.9841      4792.6 examples/second
[Step=100]      Loss=0.0519     acc=0.9842      12345.4 examples/second
[Step=150]      Loss=0.0524     acc=0.9833      12372.6 examples/second
Test Loss=0.3260, Test acc=0.9141

Epoch: 5
[Step=50]       Loss=0.0617     acc=0.9802      6315.3 examples/second
[Step=100]      Loss=0.0630     acc=0.9797      13431.7 examples/second
[Step=150]      Loss=0.0614     acc=0.9797      12610.3 examples/second
Test Loss=0.3229, Test acc=0.9113

Epoch: 6
[Step=50]       Loss=0.0731     acc=0.9754      6366.3 examples/second
[Step=100]      Loss=0.0719     acc=0.9758      12640.7 examples/second
[Step=150]      Loss=0.0718     acc=0.9759      12408.8 examples/second
Test Loss=0.3234, Test acc=0.9099

Epoch: 7
[Step=50]       Loss=0.0885     acc=0.9702      6106.5 examples/second
[Step=100]      Loss=0.0860     acc=0.9717      12448.0 examples/second
[Step=150]      Loss=0.0872     acc=0.9708      12239.8 examples/second
Test Loss=0.3256, Test acc=0.9068

Epoch: 8
[Step=50]       Loss=0.1173     acc=0.9610      6134.2 examples/second
[Step=100]      Loss=0.1194     acc=0.9599      12445.5 examples/second
[Step=150]      Loss=0.1173     acc=0.9605      12471.5 examples/second
Test Loss=0.3249, Test acc=0.9005

Epoch: 9
[Step=50]       Loss=0.1843     acc=0.9363      6212.4 examples/second
[Step=100]      Loss=0.1780     acc=0.9388      11804.6 examples/second
[Step=150]      Loss=0.1724     acc=0.9407      10877.9 examples/second
Test Loss=0.3410, Test acc=0.8903

Epoch: 10
[Step=50]       Loss=0.1532     acc=0.9495      6142.6 examples/second
[Step=100]      Loss=0.1586     acc=0.9470      13074.3 examples/second
[Step=150]      Loss=0.1582     acc=0.9464      11776.5 examples/second
Test Loss=0.3339, Test acc=0.8934
Saving…
```

```
Epoch: 11
[Step=50]       Loss=0.1566     acc=0.9456      6226.4 examples/second
[Step=100]      Loss=0.1532     acc=0.9479      11704.4 examples/second
[Step=150]      Loss=0.1514     acc=0.9485      11097.2 examples/second
Test Loss=0.3300, Test acc=0.8946
Saving…

Epoch: 12
[Step=50]       Loss=0.1441     acc=0.9504      5309.6 examples/second
[Step=100]      Loss=0.1457     acc=0.9515      12221.5 examples/second
[Step=150]      Loss=0.1454     acc=0.9513      12054.7 examples/second
Test Loss=0.3273, Test acc=0.8949
Saving…

Epoch: 13
[Step=50]       Loss=0.1325     acc=0.9542      6226.3 examples/second
[Step=100]      Loss=0.1376     acc=0.9525      12487.7 examples/second
[Step=150]      Loss=0.1391     acc=0.9525      12441.0 examples/second
Test Loss=0.3229, Test acc=0.8970
Saving…

Epoch: 14
[Step=50]       Loss=0.1314     acc=0.9566      6036.0 examples/second
[Step=100]      Loss=0.1372     acc=0.9543      11868.5 examples/second
[Step=150]      Loss=0.1358     acc=0.9542      11846.7 examples/second
Test Loss=0.3231, Test acc=0.8969

Epoch: 15
[Step=50]       Loss=0.1355     acc=0.9521      6031.4 examples/second
[Step=100]      Loss=0.1347     acc=0.9539      12181.6 examples/second
[Step=150]      Loss=0.1353     acc=0.9537      12342.5 examples/second
Test Loss=0.3204, Test acc=0.8985
Saving…

Epoch: 16
[Step=50]       Loss=0.1276     acc=0.9568      5985.7 examples/second
[Step=100]      Loss=0.1318     acc=0.9554      12347.7 examples/second
[Step=150]      Loss=0.1319     acc=0.9547      11193.9 examples/second
Test Loss=0.3189, Test acc=0.8984

Epoch: 17
[Step=50]       Loss=0.1329     acc=0.9560      6316.6 examples/second
[Step=100]      Loss=0.1312     acc=0.9562      12470.3 examples/second
[Step=150]      Loss=0.1286     acc=0.9568      12600.0 examples/second
Test Loss=0.3184, Test acc=0.8981

Epoch: 18
[Step=50]       Loss=0.1268     acc=0.9566      6341.9 examples/second
```

```
[Step=100]      Loss=0.1260     acc=0.9570      12705.8 examples/second
[Step=150]      Loss=0.1268     acc=0.9572      10798.4 examples/second
Test Loss=0.3182, Test acc=0.8986
Saving…

Epoch: 19
[Step=50]       Loss=0.1266     acc=0.9572      5950.9 examples/second
[Step=100]      Loss=0.1259     acc=0.9574      11822.7 examples/second
[Step=150]      Loss=0.1261     acc=0.9572      12023.3 examples/second
Test Loss=0.3178, Test acc=0.8985
```

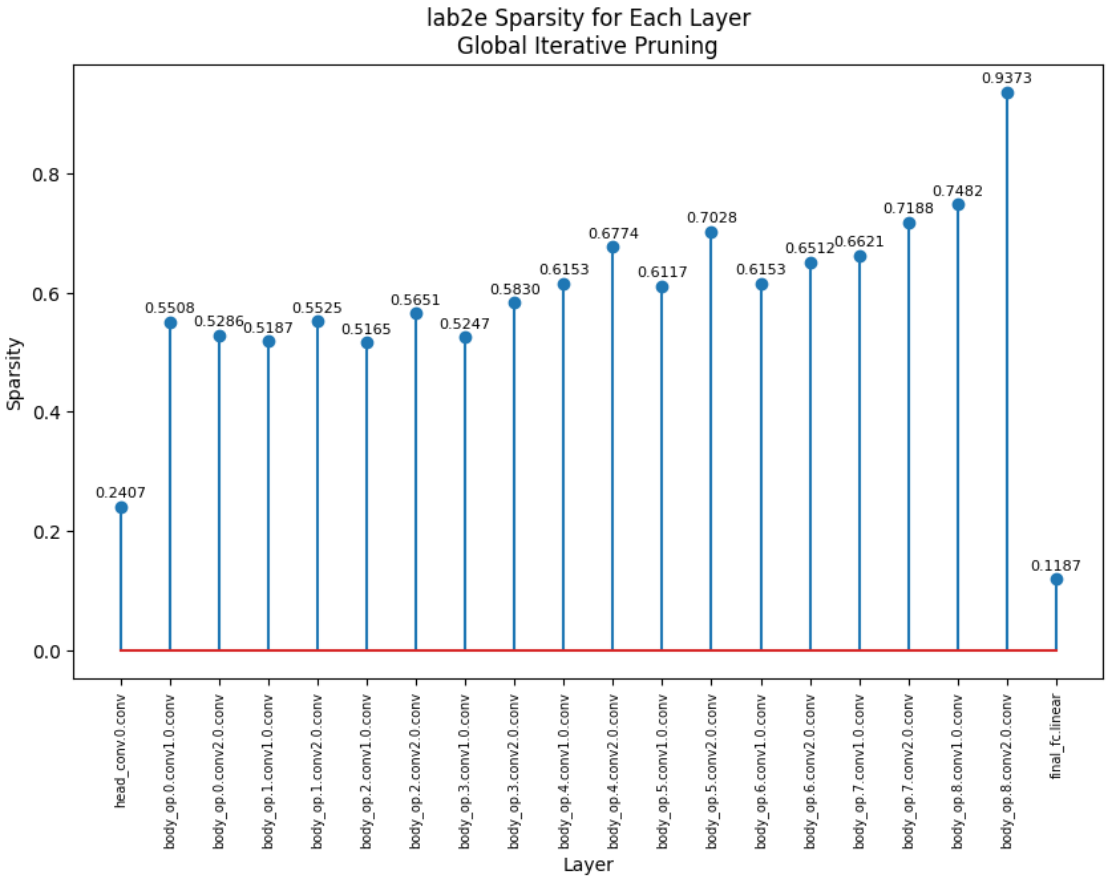
[23]:
```
net.load_state_dict(torch.load("net_after_global_iterative_prune.pt"))
sparsity_list = []
x_ticks_labels = []
zeros_sum = 0
total_sum = 0
for name, layer in net.named_modules():
    if (isinstance(layer, nn.Conv2d) or isinstance(layer, nn.Linear)) and␣
 ↪'id_mapping' not in name:
        # Your code here:
        # Convert the weight of "layer" to numpy array
        np_weight = layer.weight.detach().cpu().numpy()
        # Count number of zeros
        zeros = sum((np_weight == 0).flatten())
        # Count number of parameters
        total = len(np_weight.flatten())
        zeros_sum += zeros
        total_sum += total
        sparsity_list.append(zeros / (total / 1.0))
        x_ticks_labels.append(name)
        print('Sparsity of %s: %g' % (name, zeros / (total / 1.0)))
print('Total sparsity: %g' % (zeros_sum / (total_sum / 1.0)))
test(net)
```

```
Sparsity of head_conv.0.conv: 0.240741
Sparsity of body_op.0.conv1.0.conv: 0.550781
Sparsity of body_op.0.conv2.0.conv: 0.528646
Sparsity of body_op.1.conv1.0.conv: 0.518663
Sparsity of body_op.1.conv2.0.conv: 0.552517
Sparsity of body_op.2.conv1.0.conv: 0.516493
Sparsity of body_op.2.conv2.0.conv: 0.565104
Sparsity of body_op.3.conv1.0.conv: 0.52474
Sparsity of body_op.3.conv2.0.conv: 0.583008
Sparsity of body_op.4.conv1.0.conv: 0.615343
Sparsity of body_op.4.conv2.0.conv: 0.677409
Sparsity of body_op.5.conv1.0.conv: 0.611654
Sparsity of body_op.5.conv2.0.conv: 0.702799
Sparsity of body_op.6.conv1.0.conv: 0.615343
```

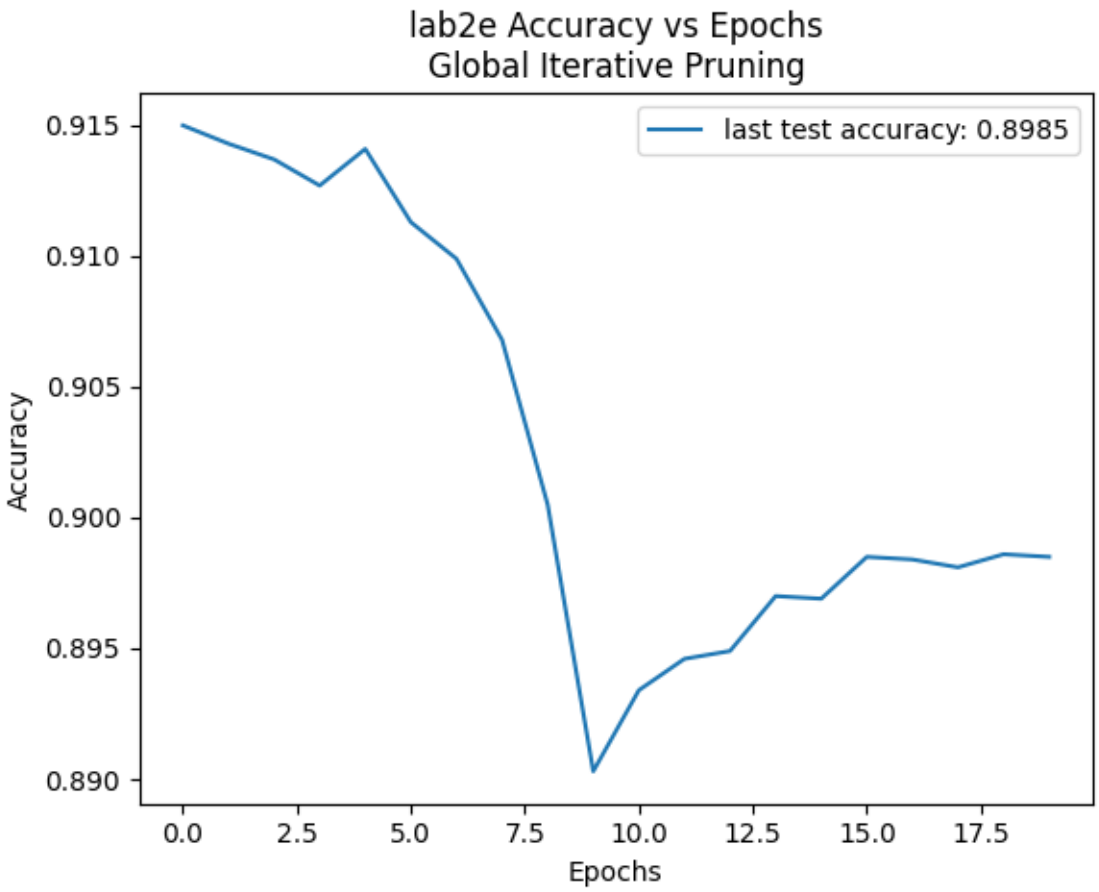
```
Sparsity of body_op.6.conv2.0.conv: 0.651232
Sparsity of body_op.7.conv1.0.conv: 0.662137
Sparsity of body_op.7.conv2.0.conv: 0.718777
Sparsity of body_op.8.conv1.0.conv: 0.74821
Sparsity of body_op.8.conv2.0.conv: 0.937283
Sparsity of final_fc.linear: 0.11875
Total sparsity: 0.699999
Files already downloaded and verified
Test Loss=0.3182, Test accuracy=0.8986
```

[23]:
```
0.8986
```

[25]:
```
fig, ax = plt.subplots(1, 1, figsize=(10, 6))
xx = range(len(sparsity_list))
ax.stem(xx, sparsity_list)
ax.set_xticks(range(len(xx)), x_ticks_labels, rotation='vertical', fontsize=7)
ax.set_xlabel('Layer')
ax.set_ylabel('Sparsity')
ax.set_title('lab2e Sparsity for Each Layer\nGlobal Iterative Pruning')
for (x, y) in zip(xx, sparsity_list):
    plt.annotate('{:.4f}'.format(y), xy=(x, y), xytext=(0, 5),␣
 ↪textcoords='offset points', ha='center', size=8)
plt.savefig('Figures/lab2e_sparsity.pdf', dpi=500, bbox_inches='tight')
```



[18]:
```
fig, ax = plt.subplots(1, 1)
xx = range(n_epochs)
ax.plot(xx, test_acc_global_prune, label='last test accuracy: %g' %␣
 ↪test_acc_global_prune[-1])
ax.set_xlabel('Epochs')
ax.set_ylabel('Accuracy')
ax.set_title('lab2e Accuracy vs Epochs\nGlobal Iterative Pruning')
ax.legend()

plt.savefig('Figures/lab2e.pdf', dpi=500, bbox_inches='tight')
```



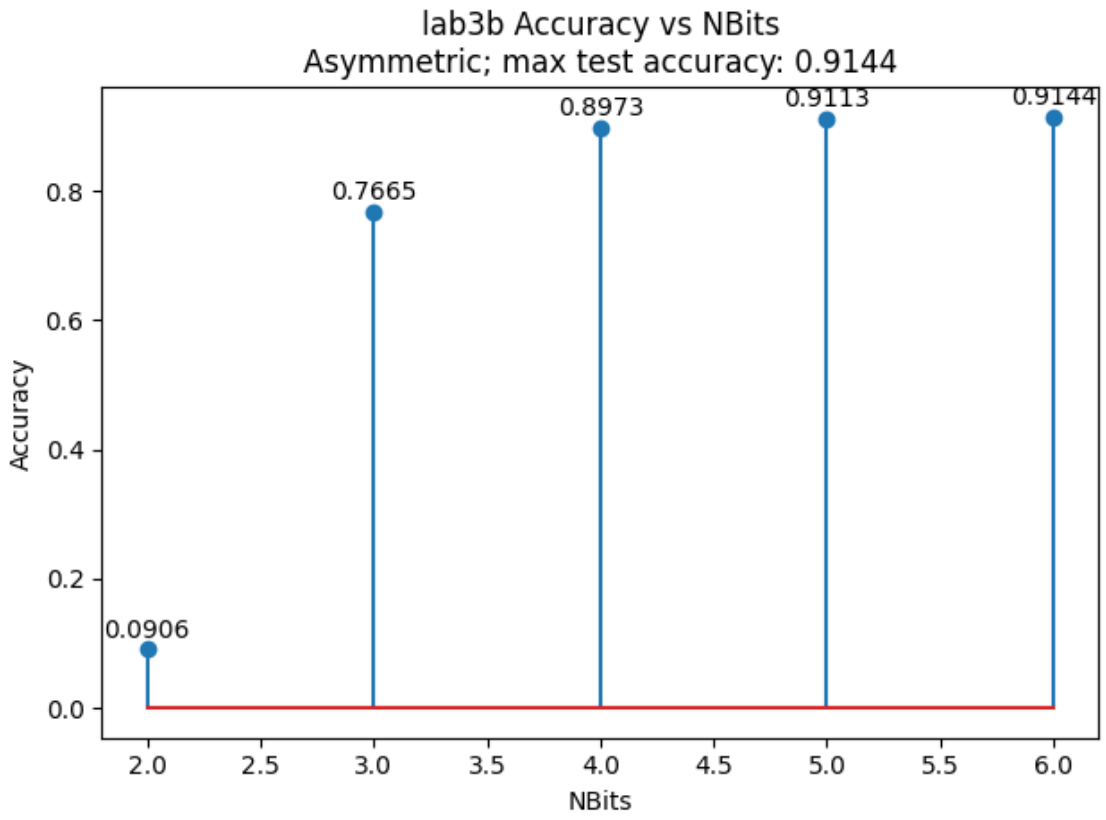
0.0.6 Lab 3 (b) and (c): Fixed-point quantization

[4]:
```
# load pretrained ResNet-20 model, report the accuracy of the floating-point␣
 ↪pretrained model.
net = ResNetCIFAR(num_layers=20)
net = net.to(device)
net.load_state_dict(torch.load("pretrained_model.pt"))
print("before quantisation:")
test(net)
print()

# Define quantized model and load weight
Nbits_arr = [2, 3, 4, 5, 6]
test_acc_lab3b = []
for iii, nb in enumerate(Nbits_arr):
    Nbits = nb #Change this value to finish (b) and (c)

    net = ResNetCIFAR(num_layers=20, Nbits=Nbits)
```

```
    net = net.to(device)
    net.load_state_dict(torch.load("pretrained_model.pt"))
    test_acc_lab3b.append(test(net))
```

```
before quantisation:
Files already downloaded and verified
Test Loss=0.3231, Test accuracy=0.9150

Files already downloaded and verified
Test Loss=9.6086, Test accuracy=0.0906
Files already downloaded and verified
Test Loss=0.9857, Test accuracy=0.7665
Files already downloaded and verified
Test Loss=0.3860, Test accuracy=0.8973
Files already downloaded and verified
Test Loss=0.3391, Test accuracy=0.9113
Files already downloaded and verified
Test Loss=0.3364, Test accuracy=0.9144
```

[10]:
```
fig, ax = plt.subplots(1, 1)
xx = Nbits_arr
ax.stem(xx, test_acc_lab3b)
# ax.plot(xx, test_acc_lab3b, label='max test accuracy: %g' % np.
 ↪max(test_acc_lab3b))
ax.set_xlabel('NBits')
ax.set_ylabel('Accuracy')
ax.set_title('lab3b Accuracy vs NBits\nAsymmetric; max test accuracy: %g' % np.
 ↪max(test_acc_lab3b))
for (x, y) in zip(xx, test_acc_lab3b):
    plt.annotate('{:.4f}'.format(y), xy=(x, y), xytext=(0, 5),␣
 ↪textcoords='offset points',ha='center')
fig.tight_layout()
plt.savefig('Figures/lab3b.pdf', dpi=500, bbox_inches='tight')
```



[11]:
```
# lab 3 part c is shown below:
```

[12]:
```
Nbits_arr_c = [2, 3, 4]
test_acc_lab3c_finetune = {}
# test accuracy for each finetuning as fn of epoch
final_test_acc_lab3c = {}  # test acc after finetuning for each Nbit
n_epochs = 20
test_acc_lab3c_b4_finetune = {} # test acc before finetuning

for iii, nb in enumerate(Nbits_arr_c):
    # Define quantized model and load weight
    Nbits = nb #Change this value to finish (b) and (c)

    net = ResNetCIFAR(num_layers=20, Nbits=Nbits)
    net = net.to(device)
    net.load_state_dict(torch.load("pretrained_model.pt"))
    test_acc_lab3c_b4_finetune[Nbits] = test(net)

    # Quantized model finetuning
    test_acc_lab3c_finetune[Nbits] = np.asarray(
        finetune(net, epochs=n_epochs, batch_size=256, lr=0.002, reg=1e-4))
```

```
    # finetune() supposedly returns list of val_accuracy values
    # Load the model with best accuracy
    net.load_state_dict(torch.load("quantized_net_after_finetune.pt"))
    final_test_acc_lab3c[Nbits] = test(net)
```

```
Files already downloaded and verified
Test Loss=9.6086, Test accuracy=0.0906
==> Preparing data..
Files already downloaded and verified
Files already downloaded and verified

Epoch: 0
[Step=50]       Loss=1.3199     acc=0.6250      5064.9 examples/second
[Step=100]      Loss=1.0535     acc=0.6870      9559.5 examples/second
[Step=150]      Loss=0.9346     acc=0.7166      9594.5 examples/second
Test Loss=0.7387, Test acc=0.7820
Saving…

Epoch: 1
[Step=200]      Loss=0.6141     acc=0.8047      3416.9 examples/second
[Step=250]      Loss=0.5912     acc=0.8040      9611.2 examples/second
[Step=300]      Loss=0.5688     acc=0.8098      9720.0 examples/second
[Step=350]      Loss=0.5459     acc=0.8157      9729.8 examples/second
Test Loss=0.6160, Test acc=0.8087
Saving…

Epoch: 2
[Step=400]      Loss=0.4761     acc=0.8359      2916.8 examples/second
[Step=450]      Loss=0.4810     acc=0.8365      8614.2 examples/second
[Step=500]      Loss=0.4718     acc=0.8390      7992.3 examples/second
[Step=550]      Loss=0.4650     acc=0.8406      9019.1 examples/second
Test Loss=0.5682, Test acc=0.8207
Saving…

Epoch: 3
[Step=600]      Loss=0.4422     acc=0.8503      2982.4 examples/second
[Step=650]      Loss=0.4296     acc=0.8514      7740.0 examples/second
[Step=700]      Loss=0.4260     acc=0.8522      8216.0 examples/second
[Step=750]      Loss=0.4242     acc=0.8532      9281.9 examples/second
Test Loss=0.5630, Test acc=0.8256
Saving…

Epoch: 4
[Step=800]      Loss=0.3994     acc=0.8660      3616.9 examples/second
[Step=850]      Loss=0.4016     acc=0.8609      8880.7 examples/second
[Step=900]      Loss=0.3993     acc=0.8597      8941.6 examples/second
[Step=950]      Loss=0.3989     acc=0.8605      8840.8 examples/second
```

```
Test Loss=0.5452, Test acc=0.8309
Saving…

Epoch: 5
[Step=1000]     Loss=0.4074     acc=0.8596      3349.6 examples/second
[Step=1050]     Loss=0.3827     acc=0.8658      9304.9 examples/second
[Step=1100]     Loss=0.3851     acc=0.8651      9023.1 examples/second
[Step=1150]     Loss=0.3811     acc=0.8667      9238.3 examples/second
Test Loss=0.5262, Test acc=0.8353
Saving…

Epoch: 6
[Step=1200]     Loss=0.3542     acc=0.8791      3387.8 examples/second
[Step=1250]     Loss=0.3542     acc=0.8769      9156.1 examples/second
[Step=1300]     Loss=0.3613     acc=0.8749      9068.4 examples/second
[Step=1350]     Loss=0.3592     acc=0.8750      9290.3 examples/second
Test Loss=0.5987, Test acc=0.8243

Epoch: 7
[Step=1400]     Loss=0.3612     acc=0.8726      3634.1 examples/second
[Step=1450]     Loss=0.3503     acc=0.8758      9067.2 examples/second
[Step=1500]     Loss=0.3517     acc=0.8771      8964.0 examples/second
[Step=1550]     Loss=0.3503     acc=0.8772      9364.0 examples/second
Test Loss=0.5400, Test acc=0.8394
Saving…

Epoch: 8
[Step=1600]     Loss=0.3455     acc=0.8785      3548.8 examples/second
[Step=1650]     Loss=0.3429     acc=0.8808      9186.2 examples/second
[Step=1700]     Loss=0.3387     acc=0.8811      9122.0 examples/second
[Step=1750]     Loss=0.3402     acc=0.8801      9614.7 examples/second
Test Loss=0.5119, Test acc=0.8431
Saving…

Epoch: 9
[Step=1800]     Loss=0.3359     acc=0.8837      3562.8 examples/second
[Step=1850]     Loss=0.3359     acc=0.8843      9036.8 examples/second
[Step=1900]     Loss=0.3303     acc=0.8847      9079.4 examples/second
[Step=1950]     Loss=0.3288     acc=0.8851      9645.1 examples/second
Test Loss=0.5415, Test acc=0.8349

Epoch: 10
[Step=2000]     Loss=0.3316     acc=0.8798      3638.5 examples/second
[Step=2050]     Loss=0.3348     acc=0.8805      9018.6 examples/second
[Step=2100]     Loss=0.3331     acc=0.8820      9029.8 examples/second
[Step=2150]     Loss=0.3294     acc=0.8833      9577.1 examples/second
Test Loss=0.5093, Test acc=0.8420
```

```
Epoch: 11
[Step=2200]     Loss=0.3162     acc=0.8890      3573.9 examples/second
[Step=2250]     Loss=0.3176     acc=0.8871      8949.2 examples/second
[Step=2300]     Loss=0.3180     acc=0.8883      9003.6 examples/second
[Step=2350]     Loss=0.3155     acc=0.8886      9855.7 examples/second
Test Loss=0.4624, Test acc=0.8555
Saving…

Epoch: 12
[Step=2400]     Loss=0.3081     acc=0.8923      3372.2 examples/second
[Step=2450]     Loss=0.3098     acc=0.8915      9674.6 examples/second
[Step=2500]     Loss=0.3116     acc=0.8908      9699.1 examples/second
Test Loss=0.5014, Test acc=0.8462

Epoch: 13
[Step=2550]     Loss=0.2429     acc=0.9199      3746.8 examples/second
[Step=2600]     Loss=0.3055     acc=0.8939      8866.0 examples/second
[Step=2650]     Loss=0.3042     acc=0.8935      9083.3 examples/second
[Step=2700]     Loss=0.3060     acc=0.8927      9076.7 examples/second
Test Loss=0.4841, Test acc=0.8508

Epoch: 14
[Step=2750]     Loss=0.2795     acc=0.9049      3611.9 examples/second
[Step=2800]     Loss=0.2960     acc=0.8965      8951.2 examples/second
[Step=2850]     Loss=0.2997     acc=0.8952      8979.6 examples/second
[Step=2900]     Loss=0.2992     acc=0.8955      9206.7 examples/second
Test Loss=0.4598, Test acc=0.8545

Epoch: 15
[Step=2950]     Loss=0.2990     acc=0.8902      3626.7 examples/second
[Step=3000]     Loss=0.2943     acc=0.8950      9052.5 examples/second
[Step=3050]     Loss=0.2922     acc=0.8971      8802.7 examples/second
[Step=3100]     Loss=0.2892     acc=0.8976      8801.3 examples/second
Test Loss=0.4655, Test acc=0.8568
Saving…

Epoch: 16
[Step=3150]     Loss=0.2909     acc=0.8970      3331.8 examples/second
[Step=3200]     Loss=0.2963     acc=0.8968      8891.1 examples/second
[Step=3250]     Loss=0.2917     acc=0.8972      8881.5 examples/second
[Step=3300]     Loss=0.2872     acc=0.8978      9036.9 examples/second
Test Loss=0.4836, Test acc=0.8509

Epoch: 17
[Step=3350]     Loss=0.2757     acc=0.9028      3560.6 examples/second
[Step=3400]     Loss=0.2887     acc=0.8952      8938.8 examples/second
[Step=3450]     Loss=0.2872     acc=0.8962      8840.4 examples/second
[Step=3500]     Loss=0.2860     acc=0.8984      8871.0 examples/second
```

```
Test Loss=0.4483, Test acc=0.8576
Saving…

Epoch: 18
[Step=3550]     Loss=0.2802     acc=0.9034      3519.2 examples/second
[Step=3600]     Loss=0.2809     acc=0.9002      9101.0 examples/second
[Step=3650]     Loss=0.2837     acc=0.8998      8806.1 examples/second
[Step=3700]     Loss=0.2874     acc=0.8980      8418.4 examples/second
Test Loss=0.4596, Test acc=0.8576

Epoch: 19
[Step=3750]     Loss=0.2893     acc=0.8980      3245.6 examples/second
[Step=3800]     Loss=0.2876     acc=0.8983      9225.6 examples/second
[Step=3850]     Loss=0.2844     acc=0.8989      8938.6 examples/second
[Step=3900]     Loss=0.2842     acc=0.8997      9348.4 examples/second
Test Loss=0.5014, Test acc=0.8440
Files already downloaded and verified
Test Loss=0.4483, Test accuracy=0.8576
Files already downloaded and verified
Test Loss=0.9857, Test accuracy=0.7665
==> Preparing data..
Files already downloaded and verified
Files already downloaded and verified

Epoch: 0
[Step=50]       Loss=0.1696     acc=0.9402      5790.4 examples/second
[Step=100]      Loss=0.1587     acc=0.9446      9041.2 examples/second
[Step=150]      Loss=0.1515     acc=0.9473      9184.6 examples/second
Test Loss=0.3917, Test acc=0.8957
Saving…

Epoch: 1
[Step=200]      Loss=0.1164     acc=0.9521      3681.9 examples/second
[Step=250]      Loss=0.1214     acc=0.9543      8930.0 examples/second
[Step=300]      Loss=0.1235     acc=0.9542      8967.8 examples/second
[Step=350]      Loss=0.1225     acc=0.9551      8787.4 examples/second
Test Loss=0.3819, Test acc=0.8987
Saving…

Epoch: 2
[Step=400]      Loss=0.1068     acc=0.9614      3329.4 examples/second
[Step=450]      Loss=0.1119     acc=0.9591      9067.3 examples/second
[Step=500]      Loss=0.1118     acc=0.9593      8954.9 examples/second
[Step=550]      Loss=0.1118     acc=0.9597      9145.0 examples/second
Test Loss=0.3750, Test acc=0.9002
Saving…

Epoch: 3
```

```
[Step=600]      Loss=0.1122     acc=0.9606      3373.3 examples/second
[Step=650]      Loss=0.1050     acc=0.9624      8948.5 examples/second
[Step=700]      Loss=0.1057     acc=0.9623      8616.6 examples/second
[Step=750]      Loss=0.1058     acc=0.9619      8644.7 examples/second
Test Loss=0.3728, Test acc=0.9002

Epoch: 4
[Step=800]      Loss=0.0996     acc=0.9629      3277.9 examples/second
[Step=850]      Loss=0.1045     acc=0.9621      9003.6 examples/second
[Step=900]      Loss=0.1031     acc=0.9628      8842.6 examples/second
[Step=950]      Loss=0.1047     acc=0.9622      8735.5 examples/second
Test Loss=0.3686, Test acc=0.9040
Saving…

Epoch: 5
[Step=1000]     Loss=0.0925     acc=0.9656      3289.7 examples/second
[Step=1050]     Loss=0.1019     acc=0.9615      9219.4 examples/second
[Step=1100]     Loss=0.1018     acc=0.9618      8997.6 examples/second
[Step=1150]     Loss=0.1023     acc=0.9624      9011.6 examples/second
Test Loss=0.3672, Test acc=0.9016

Epoch: 6
[Step=1200]     Loss=0.0971     acc=0.9681      3603.5 examples/second
[Step=1250]     Loss=0.1015     acc=0.9638      9218.9 examples/second
[Step=1300]     Loss=0.1008     acc=0.9639      8944.5 examples/second
[Step=1350]     Loss=0.1011     acc=0.9642      9182.3 examples/second
Test Loss=0.3728, Test acc=0.9010

Epoch: 7
[Step=1400]     Loss=0.0914     acc=0.9669      3034.5 examples/second
[Step=1450]     Loss=0.0907     acc=0.9675      8991.2 examples/second
[Step=1500]     Loss=0.0951     acc=0.9664      9560.0 examples/second
[Step=1550]     Loss=0.0975     acc=0.9653      9428.4 examples/second
Test Loss=0.3679, Test acc=0.9015

Epoch: 8
[Step=1600]     Loss=0.0983     acc=0.9641      3363.7 examples/second
[Step=1650]     Loss=0.0958     acc=0.9653      8885.9 examples/second
[Step=1700]     Loss=0.0964     acc=0.9651      8842.4 examples/second
[Step=1750]     Loss=0.0968     acc=0.9654      8737.4 examples/second
Test Loss=0.3743, Test acc=0.9012

Epoch: 9
[Step=1800]     Loss=0.0962     acc=0.9652      3242.6 examples/second
[Step=1850]     Loss=0.0936     acc=0.9663      8963.5 examples/second
[Step=1900]     Loss=0.0942     acc=0.9661      8960.3 examples/second
[Step=1950]     Loss=0.0935     acc=0.9664      8973.6 examples/second
Test Loss=0.3675, Test acc=0.9027
```

```
Epoch: 10
[Step=2000]     Loss=0.0971     acc=0.9646      3627.0 examples/second
[Step=2050]     Loss=0.0966     acc=0.9650      9085.6 examples/second
[Step=2100]     Loss=0.0963     acc=0.9651      9032.3 examples/second
[Step=2150]     Loss=0.0952     acc=0.9654      9184.7 examples/second
Test Loss=0.3578, Test acc=0.9034

Epoch: 11
[Step=2200]     Loss=0.0865     acc=0.9691      3356.4 examples/second
[Step=2250]     Loss=0.0929     acc=0.9664      8594.0 examples/second
[Step=2300]     Loss=0.0932     acc=0.9669      8620.8 examples/second
[Step=2350]     Loss=0.0938     acc=0.9665      8909.3 examples/second
Test Loss=0.3577, Test acc=0.9046
Saving…

Epoch: 12
[Step=2400]     Loss=0.0869     acc=0.9689      3574.0 examples/second
[Step=2450]     Loss=0.0901     acc=0.9679      8848.3 examples/second
[Step=2500]     Loss=0.0898     acc=0.9684      8947.4 examples/second
Test Loss=0.3648, Test acc=0.9022

Epoch: 13
[Step=2550]     Loss=0.0755     acc=0.9766      3390.1 examples/second
[Step=2600]     Loss=0.0876     acc=0.9686      8889.5 examples/second
[Step=2650]     Loss=0.0884     acc=0.9683      8928.4 examples/second
[Step=2700]     Loss=0.0886     acc=0.9682      9048.0 examples/second
Test Loss=0.3537, Test acc=0.9053
Saving…

Epoch: 14
[Step=2750]     Loss=0.0935     acc=0.9642      3611.2 examples/second
[Step=2800]     Loss=0.0906     acc=0.9672      9169.0 examples/second
[Step=2850]     Loss=0.0898     acc=0.9681      8868.0 examples/second
[Step=2900]     Loss=0.0891     acc=0.9684      8897.3 examples/second
Test Loss=0.3601, Test acc=0.9052

Epoch: 15
[Step=2950]     Loss=0.0934     acc=0.9652      3546.7 examples/second
[Step=3000]     Loss=0.0898     acc=0.9673      9089.0 examples/second
[Step=3050]     Loss=0.0890     acc=0.9676      8988.5 examples/second
[Step=3100]     Loss=0.0889     acc=0.9679      9194.3 examples/second
Test Loss=0.3632, Test acc=0.9032

Epoch: 16
[Step=3150]     Loss=0.0829     acc=0.9696      3325.5 examples/second
[Step=3200]     Loss=0.0890     acc=0.9676      9060.3 examples/second
[Step=3250]     Loss=0.0882     acc=0.9681      9120.1 examples/second
```

```
[Step=3300]     Loss=0.0897     acc=0.9673      9176.5 examples/second
Test Loss=0.3562, Test acc=0.9032

Epoch: 17
[Step=3350]     Loss=0.0849     acc=0.9685      3356.1 examples/second
[Step=3400]     Loss=0.0883     acc=0.9688      9136.2 examples/second
[Step=3450]     Loss=0.0896     acc=0.9686      9003.3 examples/second
[Step=3500]     Loss=0.0896     acc=0.9682      9184.7 examples/second
Test Loss=0.3600, Test acc=0.9016

Epoch: 18
[Step=3550]     Loss=0.0824     acc=0.9714      3551.5 examples/second
[Step=3600]     Loss=0.0878     acc=0.9692      9263.9 examples/second
[Step=3650]     Loss=0.0879     acc=0.9687      8926.6 examples/second
[Step=3700]     Loss=0.0887     acc=0.9686      9366.3 examples/second
Test Loss=0.3725, Test acc=0.9014

Epoch: 19
[Step=3750]     Loss=0.0858     acc=0.9672      3266.7 examples/second
[Step=3800]     Loss=0.0887     acc=0.9679      9136.1 examples/second
[Step=3850]     Loss=0.0847     acc=0.9702      8982.6 examples/second
[Step=3900]     Loss=0.0840     acc=0.9704      9328.0 examples/second
Test Loss=0.3572, Test acc=0.9050
Files already downloaded and verified
Test Loss=0.3537, Test accuracy=0.9053
Files already downloaded and verified
Test Loss=0.3860, Test accuracy=0.8973
==> Preparing data..
Files already downloaded and verified
Files already downloaded and verified

Epoch: 0
[Step=50]       Loss=0.0636     acc=0.9789      5830.0 examples/second
[Step=100]      Loss=0.0648     acc=0.9780      8858.1 examples/second
[Step=150]      Loss=0.0659     acc=0.9770      9042.0 examples/second
Test Loss=0.3373, Test acc=0.9088
Saving…

Epoch: 1
[Step=200]      Loss=0.0409     acc=0.9902      3393.4 examples/second
[Step=250]      Loss=0.0663     acc=0.9766      8828.5 examples/second
[Step=300]      Loss=0.0650     acc=0.9774      8885.3 examples/second
[Step=350]      Loss=0.0647     acc=0.9776      8957.7 examples/second
Test Loss=0.3343, Test acc=0.9094
Saving…

Epoch: 2
[Step=400]      Loss=0.0656     acc=0.9766      3312.1 examples/second
```

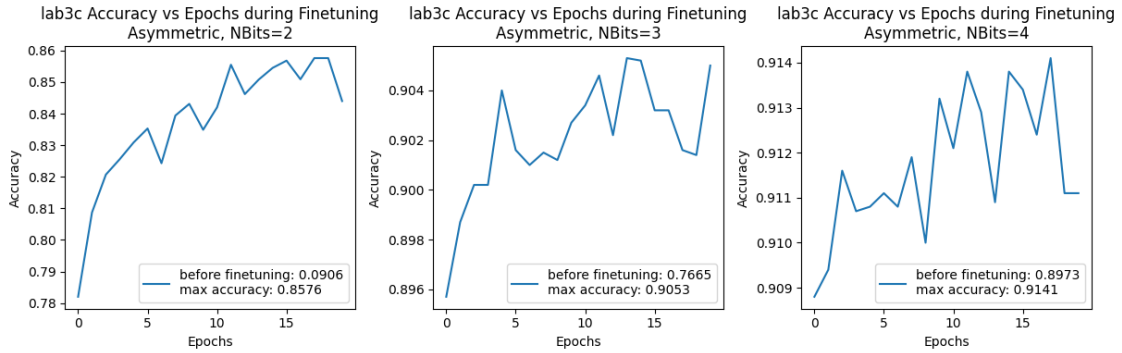
```
[Step=450]      Loss=0.0577     acc=0.9817      8889.2 examples/second
[Step=500]      Loss=0.0608     acc=0.9797      8918.0 examples/second
[Step=550]      Loss=0.0622     acc=0.9791      8865.4 examples/second
Test Loss=0.3353, Test acc=0.9116
Saving…

Epoch: 3
[Step=600]      Loss=0.0627     acc=0.9759      3325.2 examples/second
[Step=650]      Loss=0.0575     acc=0.9797      8984.9 examples/second
[Step=700]      Loss=0.0570     acc=0.9804      8935.8 examples/second
[Step=750]      Loss=0.0581     acc=0.9801      9351.3 examples/second
Test Loss=0.3322, Test acc=0.9107

Epoch: 4
[Step=800]      Loss=0.0512     acc=0.9839      3577.3 examples/second
[Step=850]      Loss=0.0567     acc=0.9816      9041.1 examples/second
[Step=900]      Loss=0.0588     acc=0.9805      8821.9 examples/second
[Step=950]      Loss=0.0588     acc=0.9804      8731.1 examples/second
Test Loss=0.3313, Test acc=0.9108

Epoch: 5
[Step=1000]     Loss=0.0583     acc=0.9824      3322.6 examples/second
[Step=1050]     Loss=0.0599     acc=0.9815      8863.7 examples/second
[Step=1100]     Loss=0.0604     acc=0.9806      8867.6 examples/second
[Step=1150]     Loss=0.0598     acc=0.9806      9044.9 examples/second
Test Loss=0.3308, Test acc=0.9111

Epoch: 6
[Step=1200]     Loss=0.0558     acc=0.9798      3600.8 examples/second
[Step=1250]     Loss=0.0556     acc=0.9813      9778.9 examples/second
[Step=1300]     Loss=0.0555     acc=0.9814      9589.5 examples/second
[Step=1350]     Loss=0.0560     acc=0.9813      9310.6 examples/second
Test Loss=0.3344, Test acc=0.9108

Epoch: 7
[Step=1400]     Loss=0.0567     acc=0.9812      3224.2 examples/second
[Step=1450]     Loss=0.0576     acc=0.9805      9086.4 examples/second
[Step=1500]     Loss=0.0577     acc=0.9804      9005.8 examples/second
[Step=1550]     Loss=0.0599     acc=0.9795      9333.0 examples/second
Test Loss=0.3323, Test acc=0.9119
Saving…

Epoch: 8
[Step=1600]     Loss=0.0539     acc=0.9816      3416.3 examples/second
[Step=1650]     Loss=0.0549     acc=0.9819      9024.3 examples/second
[Step=1700]     Loss=0.0556     acc=0.9812      9039.2 examples/second
[Step=1750]     Loss=0.0562     acc=0.9811      9628.8 examples/second
Test Loss=0.3337, Test acc=0.9100
```

```
Epoch: 9
[Step=1800]     Loss=0.0595     acc=0.9809      3369.8 examples/second
[Step=1850]     Loss=0.0582     acc=0.9813      8824.3 examples/second
[Step=1900]     Loss=0.0580     acc=0.9812      8817.4 examples/second
[Step=1950]     Loss=0.0563     acc=0.9817      9178.9 examples/second
Test Loss=0.3334, Test acc=0.9132
Saving…

Epoch: 10
[Step=2000]     Loss=0.0507     acc=0.9844      3437.9 examples/second
[Step=2050]     Loss=0.0541     acc=0.9821      8487.0 examples/second
[Step=2100]     Loss=0.0547     acc=0.9824      8808.1 examples/second
[Step=2150]     Loss=0.0553     acc=0.9820      9507.2 examples/second
Test Loss=0.3339, Test acc=0.9121

Epoch: 11
[Step=2200]     Loss=0.0524     acc=0.9823      3598.0 examples/second
[Step=2250]     Loss=0.0554     acc=0.9810      9539.4 examples/second
[Step=2300]     Loss=0.0549     acc=0.9811      9591.2 examples/second
[Step=2350]     Loss=0.0551     acc=0.9816      9555.4 examples/second
Test Loss=0.3340, Test acc=0.9138
Saving…

Epoch: 12
[Step=2400]     Loss=0.0589     acc=0.9801      3414.0 examples/second
[Step=2450]     Loss=0.0570     acc=0.9804      8979.7 examples/second
[Step=2500]     Loss=0.0568     acc=0.9806      9029.9 examples/second
Test Loss=0.3344, Test acc=0.9129

Epoch: 13
[Step=2550]     Loss=0.0447     acc=0.9844      3638.7 examples/second
[Step=2600]     Loss=0.0516     acc=0.9843      8909.1 examples/second
[Step=2650]     Loss=0.0541     acc=0.9827      8822.3 examples/second
[Step=2700]     Loss=0.0546     acc=0.9820      9170.6 examples/second
Test Loss=0.3373, Test acc=0.9109

Epoch: 14
[Step=2750]     Loss=0.0563     acc=0.9798      3613.7 examples/second
[Step=2800]     Loss=0.0537     acc=0.9818      9061.7 examples/second
[Step=2850]     Loss=0.0528     acc=0.9823      8936.2 examples/second
[Step=2900]     Loss=0.0530     acc=0.9824      8969.6 examples/second
Test Loss=0.3317, Test acc=0.9138

Epoch: 15
[Step=2950]     Loss=0.0498     acc=0.9867      3337.1 examples/second
[Step=3000]     Loss=0.0499     acc=0.9841      9156.2 examples/second
[Step=3050]     Loss=0.0522     acc=0.9831      8975.3 examples/second
```

```
[Step=3100]     Loss=0.0532     acc=0.9829      9281.6 examples/second
Test Loss=0.3345, Test acc=0.9134

Epoch: 16
[Step=3150]     Loss=0.0611     acc=0.9780      3484.5 examples/second
[Step=3200]     Loss=0.0600     acc=0.9798      9034.7 examples/second
[Step=3250]     Loss=0.0562     acc=0.9814      8955.5 examples/second
[Step=3300]     Loss=0.0552     acc=0.9818      9386.9 examples/second
Test Loss=0.3330, Test acc=0.9124

Epoch: 17
[Step=3350]     Loss=0.0602     acc=0.9803      3644.2 examples/second
[Step=3400]     Loss=0.0570     acc=0.9799      9065.9 examples/second
[Step=3450]     Loss=0.0558     acc=0.9808      8923.4 examples/second
[Step=3500]     Loss=0.0531     acc=0.9816      9067.4 examples/second
Test Loss=0.3364, Test acc=0.9141
Saving…

Epoch: 18
[Step=3550]     Loss=0.0494     acc=0.9837      3300.7 examples/second
[Step=3600]     Loss=0.0499     acc=0.9837      9015.7 examples/second
[Step=3650]     Loss=0.0513     acc=0.9833      8850.9 examples/second
[Step=3700]     Loss=0.0512     acc=0.9832      9148.4 examples/second
Test Loss=0.3413, Test acc=0.9111

Epoch: 19
[Step=3750]     Loss=0.0495     acc=0.9826      3279.0 examples/second
[Step=3800]     Loss=0.0488     acc=0.9842      9286.5 examples/second
[Step=3850]     Loss=0.0503     acc=0.9835      9694.6 examples/second
[Step=3900]     Loss=0.0513     acc=0.9828      9261.1 examples/second
Test Loss=0.3384, Test acc=0.9111
Files already downloaded and verified
Test Loss=0.3364, Test accuracy=0.9141
```

[13]:
```
fig, ax = plt.subplots(1, 3, figsize=(12, 4))

for iii, nbits in enumerate(Nbits_arr_c):
    xx = range(n_epochs)
    ax[iii].plot(xx, test_acc_lab3c_finetune[nbits], label='before finetuning:␣
 ↪%g\nmax accuracy: %g'
                 % (test_acc_lab3c_b4_finetune[nbits], np.
 ↪max(test_acc_lab3c_finetune[nbits])))
    ax[iii].set_xlabel('Epochs')
    ax[iii].set_ylabel('Accuracy')
    ax[iii].set_title('lab3c Accuracy vs Epochs during Finetuning\nAsymmetric,␣
 ↪NBits=%d' % nbits)
    ax[iii].legend()
```

```
fig.tight_layout()
plt.savefig('Figures/lab3c.pdf', dpi=500, bbox_inches='tight')
```



0.0.7 Lab3 (d) Quantize pruned model

[2]:
```
nbits_lab3d = [2, 3, 4]
test_acc_lab3d_b4_finetune = {}
test_acc_arrays_finetune = {}
test_acc_lab3d_after_finetune = {}
n_epochs = 20

for iii, nb in enumerate(nbits_lab3d):
    # Define quantized model and load weight
    Nbits = nb #Change this value to finish (d)

    net = ResNetCIFAR(num_layers=20, Nbits=Nbits)
    net = net.to(device)
    net.load_state_dict(torch.load("net_after_global_iterative_prune.pt"))
    test_acc_lab3d_b4_finetune[Nbits] = test(net)

    # Quantized model finetuning
    test_acc_arrays_finetune[Nbits] = np.array(
        finetune(net, epochs=n_epochs, batch_size=256, lr=0.002, reg=1e-4))

    # Do not load the model with best accuracy since we want the last acc value.
    # net.load_state_dict(torch.load("quantized_net_after_finetune.pt"))
    test_acc_lab3d_after_finetune[Nbits] = test(net)
```

```
Files already downloaded and verified
Test Loss=8523.9728, Test accuracy=0.1000
==> Preparing data..
Files already downloaded and verified
Files already downloaded and verified
```

```
Epoch: 0
[Step=50]       Loss=2.2584     acc=0.1487      2551.6 examples/second
[Step=100]      Loss=2.2225     acc=0.1541      4022.4 examples/second
[Step=150]      Loss=2.2040     acc=0.1603      2889.7 examples/second
Test Loss=2.1456, Test acc=0.1768
Saving…

Epoch: 1
[Step=200]      Loss=2.1437     acc=0.1777      1498.7 examples/second
[Step=250]      Loss=2.1324     acc=0.1859      5755.3 examples/second
[Step=300]      Loss=2.1203     acc=0.1960      7315.4 examples/second
[Step=350]      Loss=2.1173     acc=0.2018      6008.7 examples/second
Test Loss=2.0912, Test acc=0.2219
Saving…

Epoch: 2
[Step=400]      Loss=2.0949     acc=0.2173      2328.7 examples/second
[Step=450]      Loss=2.0796     acc=0.2308      7142.5 examples/second
[Step=500]      Loss=2.0751     acc=0.2343      6791.7 examples/second
[Step=550]      Loss=2.0707     acc=0.2361      5729.8 examples/second
Test Loss=2.0630, Test acc=0.2419
Saving…

Epoch: 3
[Step=600]      Loss=2.0576     acc=0.2409      2538.4 examples/second
[Step=650]      Loss=2.0530     acc=0.2421      7277.8 examples/second
[Step=700]      Loss=2.0493     acc=0.2451      6790.3 examples/second
[Step=750]      Loss=2.0461     acc=0.2457      6486.7 examples/second
Test Loss=2.0509, Test acc=0.2500
Saving…

Epoch: 4
[Step=800]      Loss=2.0399     acc=0.2627      2230.6 examples/second
[Step=850]      Loss=2.0310     acc=0.2558      7058.7 examples/second
[Step=900]      Loss=2.0260     acc=0.2560      6324.5 examples/second
[Step=950]      Loss=2.0229     acc=0.2574      6454.7 examples/second
Test Loss=2.0337, Test acc=0.2449

Epoch: 5
[Step=1000]     Loss=2.0018     acc=0.2617      1983.5 examples/second
[Step=1050]     Loss=2.0080     acc=0.2606      7184.5 examples/second
[Step=1100]     Loss=2.0030     acc=0.2634      6992.6 examples/second
[Step=1150]     Loss=2.0020     acc=0.2630      7130.1 examples/second
Test Loss=2.0206, Test acc=0.2597
Saving…

Epoch: 6
[Step=1200]     Loss=1.9806     acc=0.2803      1913.3 examples/second
```

```
[Step=1250]     Loss=1.9885     acc=0.2740      5945.2 examples/second
[Step=1300]     Loss=1.9872     acc=0.2716      6223.8 examples/second
[Step=1350]     Loss=1.9816     acc=0.2729      6070.4 examples/second
Test Loss=1.9910, Test acc=0.2674
Saving…

Epoch: 7
[Step=1400]     Loss=1.9636     acc=0.2856      2413.9 examples/second
[Step=1450]     Loss=1.9622     acc=0.2817      5625.7 examples/second
[Step=1500]     Loss=1.9618     acc=0.2820      6169.3 examples/second
[Step=1550]     Loss=1.9566     acc=0.2813      6561.9 examples/second
Test Loss=1.9693, Test acc=0.2750
Saving…

Epoch: 8
[Step=1600]     Loss=1.9517     acc=0.2808      2430.4 examples/second
[Step=1650]     Loss=1.9424     acc=0.2841      5413.7 examples/second
[Step=1700]     Loss=1.9401     acc=0.2849      5684.2 examples/second
[Step=1750]     Loss=1.9356     acc=0.2879      6402.2 examples/second
Test Loss=2.0433, Test acc=0.2379

Epoch: 9
[Step=1800]     Loss=1.9403     acc=0.2780      2304.2 examples/second
[Step=1850]     Loss=1.9254     acc=0.2899      5910.4 examples/second
[Step=1900]     Loss=1.9200     acc=0.2916      6326.7 examples/second
[Step=1950]     Loss=1.9147     acc=0.2928      6645.6 examples/second
Test Loss=1.9648, Test acc=0.2771
Saving…

Epoch: 10
[Step=2000]     Loss=1.8978     acc=0.2988      2007.0 examples/second
[Step=2050]     Loss=1.8933     acc=0.3024      5328.3 examples/second
[Step=2100]     Loss=1.8940     acc=0.3034      6084.0 examples/second
[Step=2150]     Loss=1.8897     acc=0.3032      5355.3 examples/second
Test Loss=1.9089, Test acc=0.2939
Saving…

Epoch: 11
[Step=2200]     Loss=1.8837     acc=0.3026      2341.1 examples/second
[Step=2250]     Loss=1.8721     acc=0.3062      6857.4 examples/second
[Step=2300]     Loss=1.8687     acc=0.3090      5330.3 examples/second
[Step=2350]     Loss=1.8647     acc=0.3116      5505.8 examples/second
Test Loss=1.9105, Test acc=0.2882

Epoch: 12
[Step=2400]     Loss=1.8482     acc=0.3193      2237.5 examples/second
[Step=2450]     Loss=1.8453     acc=0.3168      5464.0 examples/second
[Step=2500]     Loss=1.8443     acc=0.3168      5736.0 examples/second
```

```
Test Loss=1.9252, Test acc=0.2729

Epoch: 13
[Step=2550]     Loss=1.8271     acc=0.3008      2240.8 examples/second
[Step=2600]     Loss=1.8203     acc=0.3265      7001.6 examples/second
[Step=2650]     Loss=1.8133     acc=0.3278      5569.1 examples/second
[Step=2700]     Loss=1.8097     acc=0.3282      5655.1 examples/second
Test Loss=1.8428, Test acc=0.3200
Saving…

Epoch: 14
[Step=2750]     Loss=1.8031     acc=0.3340      2244.0 examples/second
[Step=2800]     Loss=1.7921     acc=0.3316      6283.3 examples/second
[Step=2850]     Loss=1.7925     acc=0.3335      6634.1 examples/second
[Step=2900]     Loss=1.7897     acc=0.3358      6687.3 examples/second
Test Loss=1.8365, Test acc=0.2960

Epoch: 15
[Step=2950]     Loss=1.7824     acc=0.3461      2300.1 examples/second
[Step=3000]     Loss=1.7727     acc=0.3393      5836.0 examples/second
[Step=3050]     Loss=1.7672     acc=0.3381      6780.5 examples/second
[Step=3100]     Loss=1.7629     acc=0.3405      6285.5 examples/second
Test Loss=1.8815, Test acc=0.2833

Epoch: 16
[Step=3150]     Loss=1.7534     acc=0.3474      2198.5 examples/second
[Step=3200]     Loss=1.7428     acc=0.3513      6244.0 examples/second
[Step=3250]     Loss=1.7403     acc=0.3530      6646.7 examples/second
[Step=3300]     Loss=1.7354     acc=0.3532      6492.4 examples/second
Test Loss=1.7803, Test acc=0.3290
Saving…

Epoch: 17
[Step=3350]     Loss=1.7069     acc=0.3707      2389.7 examples/second
[Step=3400]     Loss=1.7099     acc=0.3621      6052.0 examples/second
[Step=3450]     Loss=1.7125     acc=0.3596      5945.7 examples/second
[Step=3500]     Loss=1.7076     acc=0.3617      6578.9 examples/second
Test Loss=1.7614, Test acc=0.3492
Saving…

Epoch: 18
[Step=3550]     Loss=1.6993     acc=0.3684      2168.6 examples/second
[Step=3600]     Loss=1.6906     acc=0.3697      5199.1 examples/second
[Step=3650]     Loss=1.6898     acc=0.3688      5951.0 examples/second
[Step=3700]     Loss=1.6859     acc=0.3715      7146.8 examples/second
Test Loss=1.7413, Test acc=0.3392

Epoch: 19
```

```
[Step=3750]     Loss=1.6793     acc=0.3733      2451.1 examples/second
[Step=3800]     Loss=1.6771     acc=0.3738      7020.0 examples/second
[Step=3850]     Loss=1.6717     acc=0.3757      6958.3 examples/second
[Step=3900]     Loss=1.6672     acc=0.3772      6609.1 examples/second
Test Loss=1.7031, Test acc=0.3565
Saving…
Files already downloaded and verified
Test Loss=1.7031, Test accuracy=0.3565
Files already downloaded and verified
Test Loss=1.1891, Test accuracy=0.6081
==> Preparing data..
Files already downloaded and verified
Files already downloaded and verified

Epoch: 0
[Step=50]       Loss=1.4277     acc=0.5354      4222.7 examples/second
[Step=100]      Loss=1.1835     acc=0.6058      7022.2 examples/second
[Step=150]      Loss=1.0297     acc=0.6548      6769.1 examples/second
Test Loss=0.6359, Test acc=0.7825
Saving…

Epoch: 1
[Step=200]      Loss=0.5752     acc=0.8027      2173.6 examples/second
[Step=250]      Loss=0.5377     acc=0.8179      6475.0 examples/second
[Step=300]      Loss=0.5151     acc=0.8266      6194.4 examples/second
[Step=350]      Loss=0.4983     acc=0.8321      6088.8 examples/second
Test Loss=0.5133, Test acc=0.8250
Saving…

Epoch: 2
[Step=400]      Loss=0.4255     acc=0.8506      2096.5 examples/second
[Step=450]      Loss=0.4051     acc=0.8598      5950.1 examples/second
[Step=500]      Loss=0.3997     acc=0.8615      7218.1 examples/second
[Step=550]      Loss=0.3895     acc=0.8661      6015.9 examples/second
Test Loss=0.4873, Test acc=0.8383
Saving…

Epoch: 3
[Step=600]      Loss=0.3470     acc=0.8766      2682.0 examples/second
[Step=650]      Loss=0.3460     acc=0.8815      5688.7 examples/second
[Step=700]      Loss=0.3396     acc=0.8844      6365.3 examples/second
[Step=750]      Loss=0.3421     acc=0.8827      6453.5 examples/second
Test Loss=0.4538, Test acc=0.8501
Saving…

Epoch: 4
[Step=800]      Loss=0.3275     acc=0.8860      2154.1 examples/second
[Step=850]      Loss=0.3198     acc=0.8885      5757.1 examples/second
```

```
[Step=900]      Loss=0.3167     acc=0.8899      5449.9 examples/second
[Step=950]      Loss=0.3155     acc=0.8904      5455.2 examples/second
Test Loss=0.4395, Test acc=0.8558
Saving…

Epoch: 5
[Step=1000]     Loss=0.2984     acc=0.9020      2189.2 examples/second
[Step=1050]     Loss=0.3011     acc=0.8963      6190.7 examples/second
[Step=1100]     Loss=0.2990     acc=0.8958      7163.5 examples/second
[Step=1150]     Loss=0.2936     acc=0.8980      6157.8 examples/second
Test Loss=0.4294, Test acc=0.8606
Saving…

Epoch: 6
[Step=1200]     Loss=0.2749     acc=0.8991      1772.0 examples/second
[Step=1250]     Loss=0.2748     acc=0.9019      5588.0 examples/second
[Step=1300]     Loss=0.2736     acc=0.9026      5355.3 examples/second
[Step=1350]     Loss=0.2742     acc=0.9025      4679.5 examples/second
Test Loss=0.4084, Test acc=0.8677
Saving…

Epoch: 7
[Step=1400]     Loss=0.2616     acc=0.9081      2051.1 examples/second
[Step=1450]     Loss=0.2589     acc=0.9086      6170.7 examples/second
[Step=1500]     Loss=0.2606     acc=0.9071      5862.7 examples/second
[Step=1550]     Loss=0.2606     acc=0.9075      5641.9 examples/second
Test Loss=0.4032, Test acc=0.8680
Saving…

Epoch: 8
[Step=1600]     Loss=0.2536     acc=0.9092      2380.5 examples/second
[Step=1650]     Loss=0.2542     acc=0.9104      7170.1 examples/second
[Step=1700]     Loss=0.2512     acc=0.9118      6127.2 examples/second
[Step=1750]     Loss=0.2497     acc=0.9122      5993.7 examples/second
Test Loss=0.4039, Test acc=0.8685
Saving…

Epoch: 9
[Step=1800]     Loss=0.2454     acc=0.9147      2019.8 examples/second
[Step=1850]     Loss=0.2425     acc=0.9158      6349.9 examples/second
[Step=1900]     Loss=0.2408     acc=0.9151      6465.4 examples/second
[Step=1950]     Loss=0.2417     acc=0.9149      6120.4 examples/second
Test Loss=0.3963, Test acc=0.8713
Saving…

Epoch: 10
[Step=2000]     Loss=0.2398     acc=0.9127      2246.5 examples/second
[Step=2050]     Loss=0.2355     acc=0.9161      5403.4 examples/second
```

```
[Step=2100]     Loss=0.2376     acc=0.9152      5753.6 examples/second
[Step=2150]     Loss=0.2367     acc=0.9162      5892.1 examples/second
Test Loss=0.3939, Test acc=0.8755
Saving…

Epoch: 11
[Step=2200]     Loss=0.2332     acc=0.9165      2256.8 examples/second
[Step=2250]     Loss=0.2312     acc=0.9179      5677.7 examples/second
[Step=2300]     Loss=0.2278     acc=0.9191      6285.6 examples/second
[Step=2350]     Loss=0.2310     acc=0.9180      5419.5 examples/second
Test Loss=0.3791, Test acc=0.8772
Saving…

Epoch: 12
[Step=2400]     Loss=0.2261     acc=0.9202      2164.6 examples/second
[Step=2450]     Loss=0.2236     acc=0.9224      5809.2 examples/second
[Step=2500]     Loss=0.2241     acc=0.9219      6129.8 examples/second
Test Loss=0.3780, Test acc=0.8789
Saving…

Epoch: 13
[Step=2550]     Loss=0.1799     acc=0.9238      1804.3 examples/second
[Step=2600]     Loss=0.2177     acc=0.9222      5828.9 examples/second
[Step=2650]     Loss=0.2184     acc=0.9233      6692.2 examples/second
[Step=2700]     Loss=0.2185     acc=0.9237      6149.5 examples/second
Test Loss=0.3976, Test acc=0.8757

Epoch: 14
[Step=2750]     Loss=0.1851     acc=0.9342      1931.4 examples/second
[Step=2800]     Loss=0.2050     acc=0.9252      4914.1 examples/second
[Step=2850]     Loss=0.2103     acc=0.9252      4737.3 examples/second
[Step=2900]     Loss=0.2133     acc=0.9250      6176.9 examples/second
Test Loss=0.3798, Test acc=0.8790
Saving…

Epoch: 15
[Step=2950]     Loss=0.2011     acc=0.9301      2068.7 examples/second
[Step=3000]     Loss=0.2064     acc=0.9265      6152.8 examples/second
[Step=3050]     Loss=0.2099     acc=0.9259      5452.2 examples/second
[Step=3100]     Loss=0.2087     acc=0.9259      6601.5 examples/second
Test Loss=0.3737, Test acc=0.8813
Saving…

Epoch: 16
[Step=3150]     Loss=0.2170     acc=0.9196      2050.2 examples/second
[Step=3200]     Loss=0.2105     acc=0.9246      6915.4 examples/second
[Step=3250]     Loss=0.2071     acc=0.9266      5686.8 examples/second
[Step=3300]     Loss=0.2051     acc=0.9268      6234.8 examples/second
```

```
Test Loss=0.3775, Test acc=0.8803

Epoch: 17
[Step=3350]     Loss=0.1894     acc=0.9368      2062.4 examples/second
[Step=3400]     Loss=0.1977     acc=0.9299      7147.3 examples/second
[Step=3450]     Loss=0.2038     acc=0.9288      5848.8 examples/second
[Step=3500]     Loss=0.2009     acc=0.9296      6100.3 examples/second
Test Loss=0.3755, Test acc=0.8786

Epoch: 18
[Step=3550]     Loss=0.1927     acc=0.9318      2099.4 examples/second
[Step=3600]     Loss=0.1999     acc=0.9315      6471.2 examples/second
[Step=3650]     Loss=0.1972     acc=0.9315      5835.3 examples/second
[Step=3700]     Loss=0.1970     acc=0.9315      6041.3 examples/second
Test Loss=0.3723, Test acc=0.8828
Saving…

Epoch: 19
[Step=3750]     Loss=0.1721     acc=0.9381      2535.8 examples/second
[Step=3800]     Loss=0.1910     acc=0.9331      5669.5 examples/second
[Step=3850]     Loss=0.1923     acc=0.9328      6447.8 examples/second
[Step=3900]     Loss=0.1938     acc=0.9324      5829.4 examples/second
Test Loss=0.3688, Test acc=0.8830
Saving…
Files already downloaded and verified
Test Loss=0.3688, Test accuracy=0.8830
Files already downloaded and verified
Test Loss=0.3924, Test accuracy=0.8793
==> Preparing data..
Files already downloaded and verified
Files already downloaded and verified

Epoch: 0
[Step=50]       Loss=0.6967     acc=0.7641      4052.4 examples/second
[Step=100]      Loss=0.4982     acc=0.8292      5687.7 examples/second
[Step=150]      Loss=0.4115     acc=0.8585      5593.0 examples/second
Test Loss=0.3715, Test acc=0.8811
Saving…

Epoch: 1
[Step=200]      Loss=0.2036     acc=0.9375      2385.5 examples/second
[Step=250]      Loss=0.2031     acc=0.9301      6396.2 examples/second
[Step=300]      Loss=0.2006     acc=0.9309      6802.8 examples/second
[Step=350]      Loss=0.1965     acc=0.9321      6917.3 examples/second
Test Loss=0.3271, Test acc=0.8893
Saving…

Epoch: 2
```

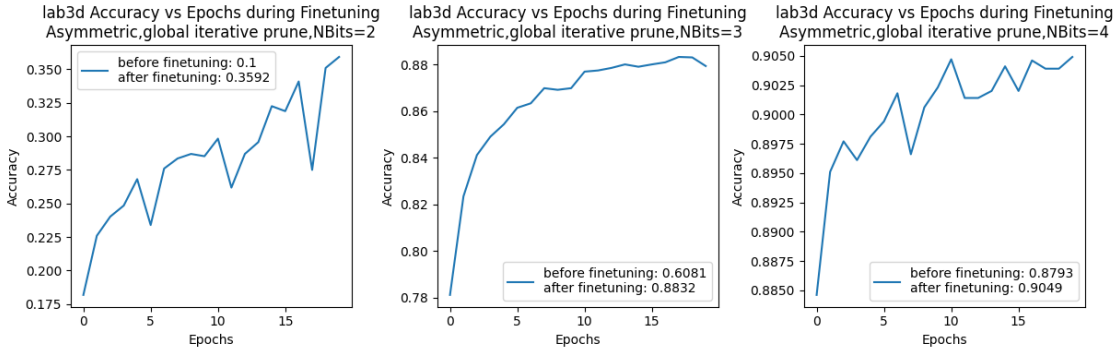
```
[Step=400]      Loss=0.1735     acc=0.9409      2093.1 examples/second
[Step=450]      Loss=0.1761     acc=0.9380      6020.2 examples/second
[Step=500]      Loss=0.1766     acc=0.9383      5815.3 examples/second
[Step=550]      Loss=0.1762     acc=0.9397      5984.7 examples/second
Test Loss=0.3157, Test acc=0.8940
Saving…

Epoch: 3
[Step=600]      Loss=0.1613     acc=0.9486      2228.8 examples/second
[Step=650]      Loss=0.1710     acc=0.9437      6017.6 examples/second
[Step=700]      Loss=0.1668     acc=0.9451      6502.6 examples/second
[Step=750]      Loss=0.1691     acc=0.9440      5935.8 examples/second
Test Loss=0.3165, Test acc=0.8964
Saving…

Epoch: 4
[Step=800]      Loss=0.1518     acc=0.9492      2282.8 examples/second
[Step=850]      Loss=0.1561     acc=0.9473      6279.8 examples/second
[Step=900]      Loss=0.1583     acc=0.9467      5865.7 examples/second
[Step=950]      Loss=0.1580     acc=0.9463      5459.3 examples/second
Test Loss=0.3109, Test acc=0.8971
Saving…

Epoch: 5
[Step=1000]     Loss=0.1585     acc=0.9467      1470.3 examples/second
[Step=1050]     Loss=0.1578     acc=0.9458      5756.0 examples/second
[Step=1100]     Loss=0.1571     acc=0.9461      6313.0 examples/second
[Step=1150]     Loss=0.1543     acc=0.9468      5814.4 examples/second
Test Loss=0.3100, Test acc=0.9001
Saving…

Epoch: 6
[Step=1200]     Loss=0.1564     acc=0.9437      2333.5 examples/second
[Step=1250]     Loss=0.1507     acc=0.9463      5865.2 examples/second
[Step=1300]     Loss=0.1472     acc=0.9477      5825.0 examples/second
[Step=1350]     Loss=0.1452     acc=0.9487      5583.1 examples/second
Test Loss=0.3526, Test acc=0.8932

Epoch: 7
[Step=1400]     Loss=0.1462     acc=0.9513      2358.5 examples/second
[Step=1450]     Loss=0.1482     acc=0.9503      6254.3 examples/second
[Step=1500]     Loss=0.1451     acc=0.9511      6041.6 examples/second
[Step=1550]     Loss=0.1449     acc=0.9509      6297.7 examples/second
Test Loss=0.3447, Test acc=0.8991

Epoch: 8
[Step=1600]     Loss=0.1276     acc=0.9556      2116.9 examples/second
[Step=1650]     Loss=0.1265     acc=0.9560      5582.5 examples/second
```

```
[Step=1700]     Loss=0.1320     acc=0.9530      5041.7 examples/second
[Step=1750]     Loss=0.1342     acc=0.9529      4866.9 examples/second
Test Loss=0.3123, Test acc=0.9019
Saving…

Epoch: 9
[Step=1800]     Loss=0.1250     acc=0.9539      2431.3 examples/second
[Step=1850]     Loss=0.1270     acc=0.9551      4829.3 examples/second
[Step=1900]     Loss=0.1276     acc=0.9543      5838.1 examples/second
[Step=1950]     Loss=0.1294     acc=0.9538      6848.1 examples/second
Test Loss=0.3368, Test acc=0.9031
Saving…

Epoch: 10
[Step=2000]     Loss=0.1238     acc=0.9568      2374.8 examples/second
[Step=2050]     Loss=0.1251     acc=0.9556      5640.0 examples/second
[Step=2100]     Loss=0.1247     acc=0.9552      6107.3 examples/second
[Step=2150]     Loss=0.1245     acc=0.9551      6087.2 examples/second
Test Loss=0.3415, Test acc=0.9003

Epoch: 11
[Step=2200]     Loss=0.1217     acc=0.9578      2327.4 examples/second
[Step=2250]     Loss=0.1211     acc=0.9566      6317.9 examples/second
[Step=2300]     Loss=0.1195     acc=0.9576      6078.1 examples/second
[Step=2350]     Loss=0.1205     acc=0.9574      6444.7 examples/second
Test Loss=0.3428, Test acc=0.9015

Epoch: 12
[Step=2400]     Loss=0.1196     acc=0.9557      2055.1 examples/second
[Step=2450]     Loss=0.1187     acc=0.9564      6098.4 examples/second
[Step=2500]     Loss=0.1199     acc=0.9563      6342.8 examples/second
Test Loss=0.3553, Test acc=0.9015

Epoch: 13
[Step=2550]     Loss=0.1105     acc=0.9629      1285.4 examples/second
[Step=2600]     Loss=0.1130     acc=0.9620      4766.6 examples/second
[Step=2650]     Loss=0.1148     acc=0.9603      5926.2 examples/second
[Step=2700]     Loss=0.1122     acc=0.9612      5831.9 examples/second
Test Loss=0.3735, Test acc=0.9012

Epoch: 14
[Step=2750]     Loss=0.1233     acc=0.9564      2233.3 examples/second
[Step=2800]     Loss=0.1131     acc=0.9602      7137.4 examples/second
[Step=2850]     Loss=0.1098     acc=0.9621      7470.3 examples/second
[Step=2900]     Loss=0.1101     acc=0.9623      6062.7 examples/second
Test Loss=0.3353, Test acc=0.9035
Saving…
```

```
Epoch: 15
[Step=2950]     Loss=0.1143     acc=0.9586      2158.8 examples/second
[Step=3000]     Loss=0.1147     acc=0.9590      6172.7 examples/second
[Step=3050]     Loss=0.1135     acc=0.9602      7081.9 examples/second
[Step=3100]     Loss=0.1115     acc=0.9610      6637.8 examples/second
Test Loss=0.3364, Test acc=0.9027

Epoch: 16
[Step=3150]     Loss=0.1184     acc=0.9570      2410.7 examples/second
[Step=3200]     Loss=0.1092     acc=0.9616      7257.2 examples/second
[Step=3250]     Loss=0.1094     acc=0.9609      7521.6 examples/second
[Step=3300]     Loss=0.1083     acc=0.9617      5921.7 examples/second
Test Loss=0.3262, Test acc=0.9011

Epoch: 17
[Step=3350]     Loss=0.1003     acc=0.9642      2214.1 examples/second
[Step=3400]     Loss=0.1039     acc=0.9633      6910.0 examples/second
[Step=3450]     Loss=0.1051     acc=0.9623      6623.9 examples/second
[Step=3500]     Loss=0.1048     acc=0.9627      6385.0 examples/second
Test Loss=0.3378, Test acc=0.9035

Epoch: 18
[Step=3550]     Loss=0.0929     acc=0.9677      2317.9 examples/second
[Step=3600]     Loss=0.0987     acc=0.9667      6866.8 examples/second
[Step=3650]     Loss=0.1022     acc=0.9648      6104.4 examples/second
[Step=3700]     Loss=0.1022     acc=0.9642      4787.4 examples/second
Test Loss=0.3430, Test acc=0.9055
Saving…

Epoch: 19
[Step=3750]     Loss=0.0954     acc=0.9674      2287.8 examples/second
[Step=3800]     Loss=0.1015     acc=0.9649      6324.0 examples/second
[Step=3850]     Loss=0.0972     acc=0.9665      5853.9 examples/second
[Step=3900]     Loss=0.0986     acc=0.9656      5434.9 examples/second
Test Loss=0.3385, Test acc=0.9028
Files already downloaded and verified
Test Loss=0.3385, Test accuracy=0.9028
```

[25]:
```
fig, ax = plt.subplots(1, 3, figsize=(12, 4))

for iii, nbits in enumerate(nbits_lab3d):
    xx = range(n_epochs)
    ax[iii].plot(xx, test_acc_arrays_finetune[nbits], label='before finetuning:␣
 ↪%g\nafter finetuning: %g'
                 % (test_acc_lab3d_b4_finetune[nbits],␣
 ↪test_acc_lab3d_after_finetune[nbits]))
    ax[iii].set_xlabel('Epochs')
```

```
    ax[iii].set_ylabel('Accuracy')
    ax[iii].set_title('lab3d Accuracy vs Epochs during Finetuning\n'
                      'Asymmetric,global iterative prune,NBits=%d' % nbits)
    ax[iii].legend()
fig.tight_layout()
plt.savefig('Figures/lab3d.pdf', dpi=500, bbox_inches='tight')
```



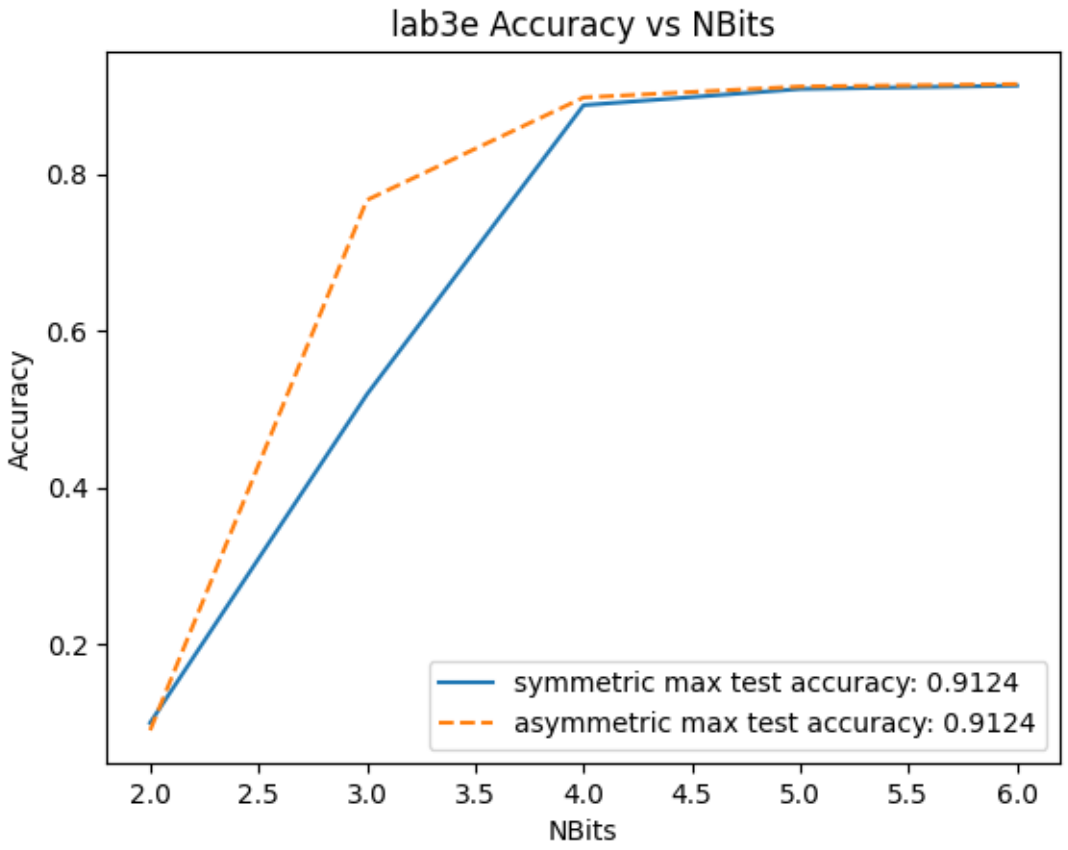
0.0.8 Lab3 (e) Symmetric quantization

Implement symmetric quantization in FP_layers.py, and repeat the process in (b)

[2]:
```
# check the performance of symmetric quantization with 6, 5, 4, 3, 2 bits
# Define quantized model and load weight
Nbits_arr_lab3e = [2, 3, 4, 5, 6]
test_acc_lab3e = []
for iii, nb in enumerate(Nbits_arr_lab3e):
    Nbits = nb #Change this value

    net = ResNetCIFAR(num_layers=20, Nbits=Nbits, symmetric=True)  # symmetric
    net = net.to(device)
    net.load_state_dict(torch.load("pretrained_model.pt"))
    test_acc_lab3e.append(test(net))
```

```
Files already downloaded and verified
Test Loss=42.7983, Test accuracy=0.1000
Files already downloaded and verified
Test Loss=2.3751, Test accuracy=0.5186
Files already downloaded and verified
Test Loss=0.4227, Test accuracy=0.8875
Files already downloaded and verified
Test Loss=0.3518, Test accuracy=0.9081
Files already downloaded and verified
Test Loss=0.3276, Test accuracy=0.9124
```

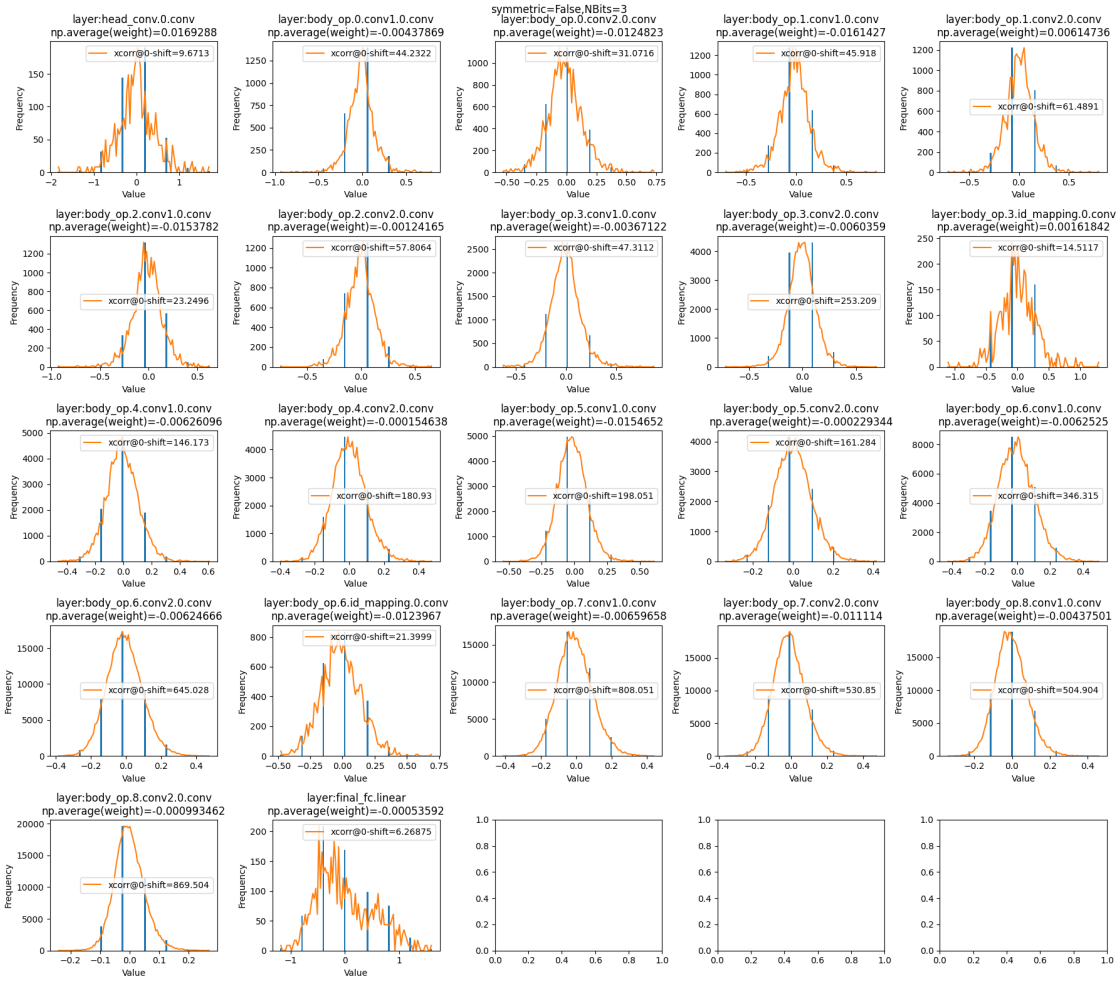
[5]:
```
fig, ax = plt.subplots(1, 1)
xx = Nbits_arr_lab3e
ax.plot(xx, test_acc_lab3e, label='symmetric max test accuracy: %g' % np.
 ↪max(test_acc_lab3e))
ax.plot(xx, test_acc_lab3b, '--', label='asymmetric max test accuracy: %g' % np.
 ↪max(test_acc_lab3e))
ax.set_xlabel('NBits')
ax.set_ylabel('Accuracy')
ax.set_title('lab3e Accuracy vs NBits')
ax.legend()

plt.savefig('Figures/lab3e.pdf', dpi=500, bbox_inches='tight')
```



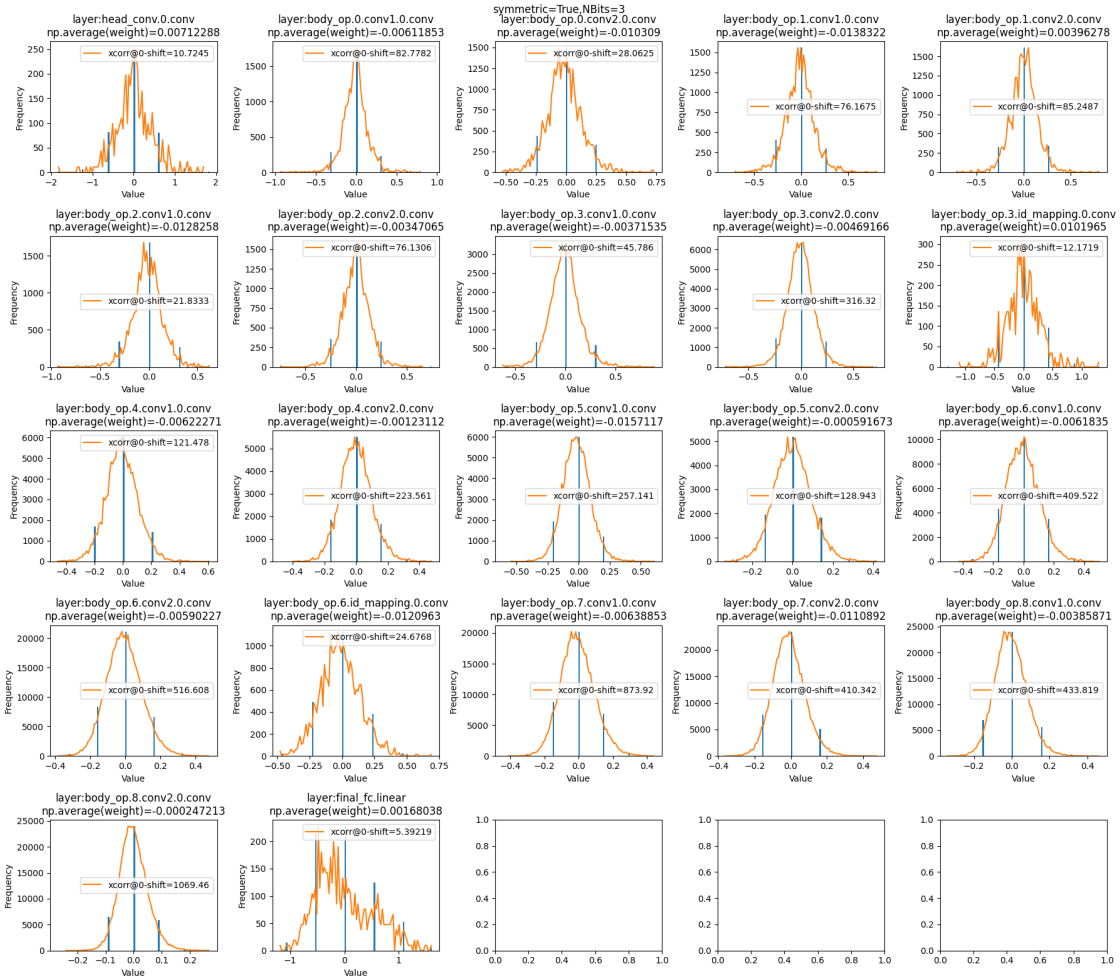
[11]:
```
def plot_quant_symmetry(symmetric, NBits=3, bins=100, save=False):

    net = ResNetCIFAR(num_layers=20) # this is for original weights
    net = net.to(device)
    net.load_state_dict(torch.load("pretrained_model.pt"))
    weights_dict_quant = {}
    w_dict_orig = {}
```

```
    conv_list = []
    for name, module in net.named_modules():
        if isinstance(module, nn.Conv2d) or isinstance(module, nn.Linear):
            w_dict_orig[str(name)] = module.weight.detach().cpu().numpy().
↪flatten()
            www = STE().forward(ctx=None, w=module.weight, bit=NBits,␣
↪symmetric=symmetric)
            np_weights = www.cpu().detach().numpy()
            weights_dict_quant[str(name)] = np_weights.flatten()
    # print(len(weights_dict_quant.keys()))
    # print(weights_dict_quant.keys())
    fig, ax = plt.subplots(5, 5, figsize=(18, 16))
    ax_flat = ax.flatten()
    for iii, layer_name in enumerate(list(weights_dict_quant.keys())):
        y_val, edges = np.histogram(w_dict_orig[layer_name], bins=bins)
        x_val = np.convolve(edges, [0.5, 0.5])[1:-1]  # compute average btw␣
↪adjacent elements of edges

        y_val_quant, _ = np.histogram(weights_dict_quant[layer_name], bins=bins)
        ax_flat[iii].hist(weights_dict_quant[layer_name], bins=bins)
        ax_flat[iii].set_title(str("layer:%s\nnp.average(weight)=%g" %
                                   (layer_name, np.
↪average(weights_dict_quant[layer_name]))))
        ax_flat[iii].set_xlabel('Value')
        ax_flat[iii].set_ylabel('Frequency')
        conv = np.convolve(y_val_quant, y_val, 'valid')
        # if len(conv) % 2 == 0:
            # zeroshift_convolve = 0.5*(conv[int((len(conv)-1)/2)] +␣
↪conv[int((len(conv)+1)/2)]) \
            # / len(weights_dict_quant[layer_name])
        # else:
            # zeroshift_convolve = (conv[int(len(conv)/2)]) /␣
↪len(weights_dict_quant[layer_name])
        zeroshift_convolve = conv / len(weights_dict_quant[layer_name])
        ax_flat[iii].plot(x_val, y_val * (np.max(y_val_quant) / np.max(y_val)),
                          label='xcorr@0-shift=%g' % zeroshift_convolve)
        ax_flat[iii].legend()
        conv_list.append(zeroshift_convolve)
    fig.suptitle('symmetric=' + str(symmetric) + ',NBits=' + str(NBits))
    fig.tight_layout()
    if save:
        plt.savefig('Figures/lab3e_hist_sym=%s_NBits=%d.pdf' % (symmetric,␣
↪NBits), dpi=500, bbox_inches='tight')
    return conv_list
```

```
c_asym = plot_quant_symmetry(symmetric=False, save=True)
```

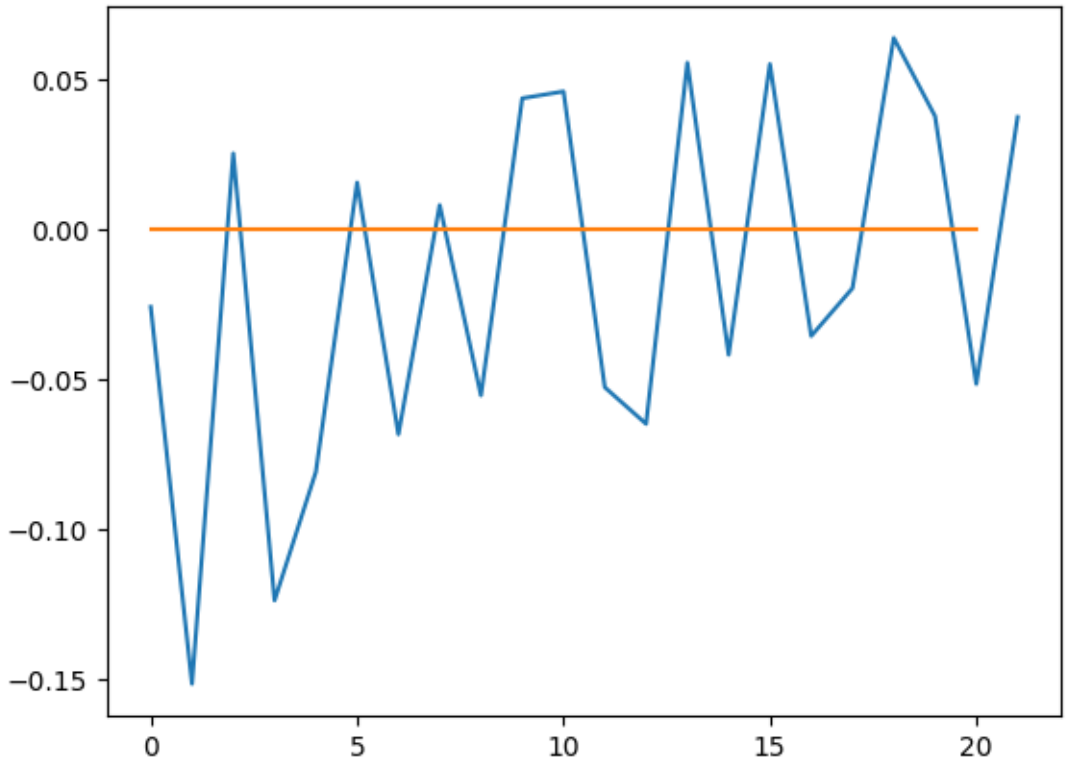


[12]:
```
c_sym = plot_quant_symmetry(symmetric=True, save=True)
```



[20]:
```
difference = np.array(c_asym) - np.array(c_sym)
plt.plot(0.5*difference / (np.array(c_asym) + np.array(c_sym)))
plt.plot([0, 20], [0, 0])
```

[20]: [<matplotlib.lines.Line2D at 0x7f6401a07fa0>]


0.05
0.00
−0.05
−0.10
−0.15
0
5
10
15
20