ECE 661 - Homework 2

Michael Li (zl310)

October 4, 2023

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

Contents

List of Figures

1 True/False Questions (30 pts)

Problem 1.1 (3 pts) Batch normalization normalizes the batch inputs by subtracting the mean, so the outputs of BN module have zero mean accordingly.

True but incomplete. Batch normalisation does normalise the batch inputs by subtracting the mean. However, a BN module placed between the output of an activation layer and the input of the next layer also provides data with zero variance, not just zero mean.

Problem 1.2 (3 pts) PyTorch provides an efficient way of tensor computation and many modularised implementation of layers. As a result, you do not necessarily need to write your own code for standard back-propagation algorithms like Adam.

True. PyTorch offers modules within `torch.nn` (e.g., `nn.Conv2d, nn.ReLU`) as the starting point for building graphs and layers. They allow back-propagation and can be stacked on top of each other to produce modularised processes. PyTorch also offers classes within `torch.optim` package for standard back-propagation algorithms. For example, `optim.Adam` and `optim.SGD` are two optimisers that allow us to perform gradient descent.

Problem 1.3 (3 pts) Data augmentation techniques are always beneficial for any kinds of CNNs and any kinds of images.

False. Data augmentation can hurt models that use a relatively small training set due to under-fitting. Some images may also be unsuitable for certain kinds of images. For example, MNIST images of 6 cannot be flipped vertically since that would become images for 9.

Problem 1.4 (3 pts) Without batch normalisation, the CNNs can hardly or at least converge very slowly during the training. This is also true for dropout.

False. CNNs are indeed very difficult to train (i.e., converge slowly) when batch normalisation isn't used. However, it is not true that the model can hardly converge without dropout, which is a regularisation technique. There are many different regularisation techniques and the main point of implementing them is to reduce the possibility of over-fitting. So, without dropout, the CNN will (probably) reach a higher training accuracy more quickly, meaning that dropout actually makes the CNN training process longer.

Problem 1.5 (3 pts) Dropout is a common technique to combat over-fitting. If L-normalisation are further incorporated at the same time, the performance can be even better.

False. On the bottom-right of slide 11 in lecture 6, it is indicated that dropout occasionally doesn't cooperate well with L-norm regularisation (L1 or L2 regularisation). As we can see in Figure 1a, the test classification error is 1.25% when dropout is used with L2 regularisation. However, the model using Dropout without L2 regularisation (Figure 1b) has the same error rate. Figure 2 also indicates that combining dropout & L-norm regularisation does worse than with just dropout. Combining dropout and another regularisation technique, max-norm, seems to be better than dropout and L2. Therefore, it's **not necessarily good practice** to combine L-norm regularisation with dropout.

Method	Test Classification error %
L2	1.62
L2 + L1 applied towards the end of training	1.60
L2 + KL-sparsity	1.55
Max-norm	1.35
Dropout + L2	1.25
Dropout + Max-norm	**1.05**

(a) Accuracy data when using dropout + something else (MNIST).

Method	Unit Type	Architecture	Error %
Standard Neural Net (Simard et al., 2003)	Logistic	2 layers, 800 units	1.60
SVM Gaussian kernel	NA	NA	1.40
Dropout NN	Logistic	3 layers, 1024 units	1.35
Dropout NN	ReLU	3 layers, 1024 units	1.25
Dropout NN + max-norm constraint	ReLU	3 layers, 1024 units	1.06
Dropout NN + max-norm constraint	ReLU	3 layers, 2048 units	1.04
Dropout NN + max-norm constraint	ReLU	2 layers, 4096 units	1.01
Dropout NN + max-norm constraint	ReLU	2 layers, 8192 units	0.95
Dropout NN + max-norm constraint (Goodfellow et al., 2013)	Maxout	2 layers, (5 × 240) units	0.94
DBN + finetuning (Hinton and Salakhutdinov, 2006)	Logistic	500-500-2000	1.18
DBM + finetuning (Salakhutdinov and Hinton, 2009)	Logistic	500-500-2000	0.96
DBN + dropout finetuning	Logistic	500-500-2000	0.92
DBM + dropout finetuning	Logistic	500-500-2000	**0.79**

(b) More accuracy data.

Figure 1: Tables 2 and 9 from Hinton, Geoffrey E., et al. "Improving neural networks by preventing co-adaptation of feature detectors." (2012)

Algorithm	*MSE*	*RMSE*	*MAE*
CNN	0.0448407	0.2117563	0.1787162
CNN-Ridge regularization	0.0195314	0.1397548	0.1040732
CNN-Lasso regularization	0.0262613	0.1620535	0.1269035
CNN-EN regularization	**0.0129773**	**0.1139183**	**0.0748202**
CNN-Dropout	0.0213447	0.1162847	0.1133612
CNN-Dropout-Ridge regularization	0.0396523	0.1991288	0.1664986
CNN-Dropout-Lasso regularization	0.0252781	0.1589910	0.1241461
CNN-Dropout-EN regularization	0.0202722	0.1423805	0.1084357

Figure 2: Combining dropout with other regularisation techniques. Table III from Z. Farhadi, H. Bevrani and M. -R. Feizi-Derakhshi, "Combining Regularization and Dropout Techniques for Deep Convolutional Neural Network".

Problem 1.6 (3 pts) During training, the Lasso (L1) regulariser makes the model to have a higher sparsity compared to Ridge (L2) regulariser.

True. According to Figure 3, the level sets (level set: the set of points in the domain of a function where the function is constant, i.e., weight values for which $\sum_i |w_i|$ is constant) of the L1 regulariser have corners, whereas the level set of the L2 regulariser is generally round. Given that the level sets of the loss function generally do not have sharp corners, it is more likely for the loss function's level set to touch the level set of the L1 regulariser at a corner. This means that more weight values are zero when L1 is used and L1 results in sparser models.

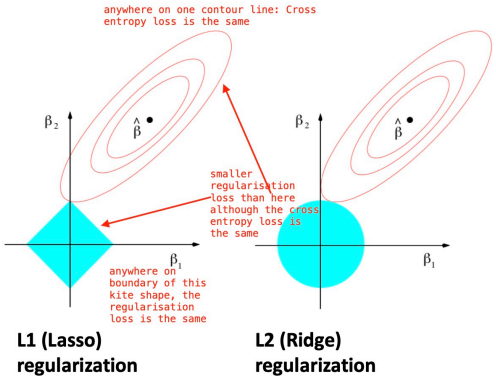


Figure 3: L1 vs L2 regularisation.

Problem 1.7 (3 pts) Though leaky ReLU solves the problem of dead neurons compared to vanilla ReLU, it could makes training unstable.

True. According to lecture 5 slide 17: "the different slopes for the negative and positive halves make the training process of some neural network architectures unstable. Under most cases, using leaky ReLU may take more time to reach convergence." Leaky ReLU also introduces another hyper-parameter, i.e., the slope of the negative section, into the model. This means that the model performance may vary depending on what the value of this hyper-parameter is.

Problem 1.8 (3 pts) MobileNets use depth-wise separable convolution to improve the model efficiency. If we replace all of the 3x3 convolution layers to 3x3 depth-wise separable convolution layers in ResNet architectures, we are likely to observe approximately 9x speedup for these layers.

Approximately **True**. Depth-wise separable convolution first computes the depth-wise 2D convolution for each channel with each kernel only convolving over a singular 2D channel (no channel cross-talk at this stage), which requires $D_k \times D_k \times M \times D_F \times D_F$ MACs. Then, the second step involves computing point-wise 1×1 convolutions to allow the number of channels to change, which requires $1 \times 1 \times D_F \times D_F \times M \times N$ MACs. D_k is the kernel size of the depth-wise convolution kernels, D_F is the 2D input/output feature map size (assuming correct padding is added such that the inputs and outputs have the same feature map size), M, N are the number of input and output channels, respectively. Regular 3×3 convolutions have $3 \times 3 \times M \times N \times D_F \times D_F$ MACs (assuming the feature map shapes are the same for the input and output). So, the ratio of MACs required for regular convolution to MACs required for depth-wise separable convolution is $\frac{3\times3\times M\times N\times D_F\times D_F}{3\times3\times M\times D_F\times D_F+1\times1\times D_F\times D_F\times M\times N} = \frac{3\times3}{\frac{3\times3}{N}+1}$. For sufficiently large input channel number (N), we use approximately 9 times more MACs when using regular convolution. In the case of ResNet architectures with large feature maps, we speed up by a factor of 9 when using the depth-wise separable convolution because the number of output channels (N) ranges from 64 ($\frac{3\times3}{\frac{3\times3}{64}+1} \approx 7.89$) to 512 ($\frac{3\times3}{\frac{3\times3}{512}+1} \approx 8.84$). If we aggressively round, then there is approximately a 9x speed-up.

Problem 1.9 (3 pts) To achieve fewer parameters than early CNN designs (e.g., AlexNet) while maintaining comparable performance, SqueezeNet puts most of the computations in the later stage of the CNN design.

False. SqueezeNet down-samples late in the network so that more MACs can be used on larger activation maps which are at the start of the network. This means that the computations are spent in earlier stages of the CNN where the activation maps are larger and not in later stages where the activation maps have already been down-sampled.

Problem 1.10 (3 pts) The shortcut connections in ResNets result in smoother loss surface.

True. According to Figure 1 in this paper, the loss surface is indeed smoother when there are shortcut connections.

2 Lab (1): Training SimpleNN for CIFAR-10 classification (15+4 pts)

(a) (2 pts) As a sanity check, we should verify the implementation of the SimpleNN model at Step 0. How can you check whether the model is implemented correctly?
Hint: 1) Consider creating dummy inputs that are of the same size as CIFAR-10 images, passing them through the model, and see if the model's outputs are of the correct shape. 2) Count the total number of parameters of all conv/FC layers and see if it meets your expectation.

Let's consider the shapes of the inputs and outputs to each layer (no padding is applied, stride for convolutions is 1, stride for max pooling is 2):

- input (x): shape is $(3, 32, 32)$, i.e. three channels with 32×32 spatial size.
- output of conv1/input to ReLU and max pool after conv1: $(8, 28, 28)$ where $W_2 = H_2 = \frac{32-5+2\times0}{1} + 1$, given that $K = 5, P = 0, S = 1$.
- (after first ReLU, which doesn't change the shape of its input) output of first max pooling/input to conv2: $(8, 14, 14)$ given that pooling stride $S = 2$.
- output of conv2/input to ReLU and max pooling after conv2: $(16, 12, 12)$ where $W_2 = H_2 = \frac{14-3+2\times0}{1} + 1 = 12$ given that $K = 3, P = 0, S = 1$.
- (after second ReLU, which doesn't change the shape of its input) output of second max pool/input to flattening layer: $(16, 6, 6)$ since $S = 2$ for max pool.
- output of flattening layer/input to FC1: $16 \times 6 \times 6 = 576$, which matches up with the line `self.fc1 = nn.Linear(16*6*6, 120)`
- output of FC1/input to FC2: 120 channels as specified by the line `self.fc1 = nn.Linear(16*6*6, 120)`
- output of FC2/input to FC3: 84 channels according to `self.fc2 = nn.Linear(120, 84`
- otuput of FC3 (final output): 10 channels according to `self.fc3 = nn.Linear(84, 10)`

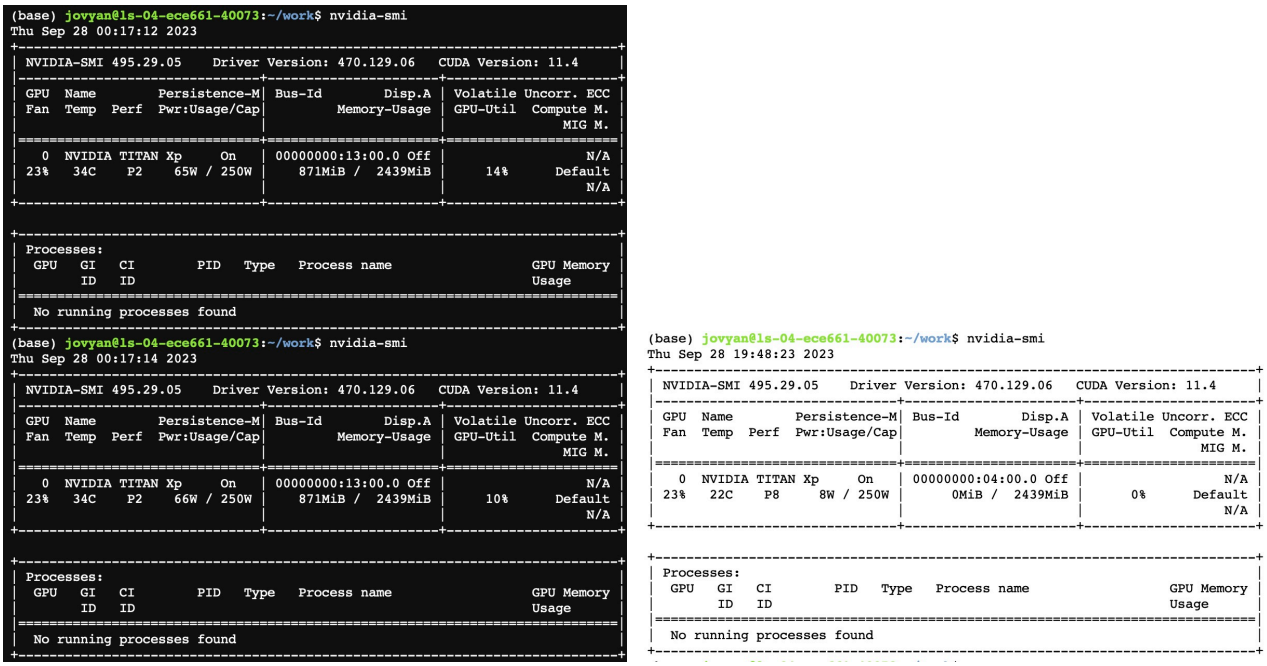
In the PDF report, give a brief description on how the code helps you know that SimpleNN is implemented correctly.
I created a test input with three channels and a size of 32×32, which has exactly the same shape as one of the RGB images in the `CIFAR10` dataset. If I run this test input through the model's forward function, the output shape is 1×10 (`torch.Size([1, 10])`). Since the output shape is what I expect when I provide an input of the correct shape, I know that the model has the correct parameters. Please see step 0 of simplenn-cifar10.ipynb for the implementation.

(b) (2 pts) Data preprocessing is crucial to enable successful training and inference of DNN models. Specify the preprocessing functions at Step 1 and briefly discuss what operations you use and why.

The two pre-processing steps I used are: `transforms.ToTensor()` and `transforms.Normalize(`μ`, `σ`)`. The normalisation transformation is used because this ensures that all images that are loaded would have roughly the same variation in their values. One set of inputs might have values ranging from 0 to 255 while another set might go from 0 to 1. If both sets of inputs are not normalised with $\mu = 0, \sigma = 1$, then the CNN trained on one set of data would have weights and biases that result in very low accuracy when applied on another set of data. So, normalisation helps with generalising the model. The `ToTensor()` function was used because `transforms.Normalize(`μ`, `σ`)` does not take in PIL images which is how the data is formatted. The pre-processing steps for the training and validation sets are exactly the same. Please see step 1 in simplenn-cifar10.ipynb for more details.

(c) (2 pts) During the training, we need to feed data to the model, which requires an efficient data loading process. This is typically achieved by setting up a dataset and a dataloader. Please go to Step 2 and build the actual training/validation datasets and dataloaders. Note, instead of using the `CIFAR10` dataset class from `torchvision.datasets`, here you are asked to use our own CIFAR-10 dataset class, which is imported from `tools.dataset`. As for the dataloader, we encourage you to use `torch.utils.data.DataLoader`.
Please see step 2 of simplenn-cifar10.ipynb.

(d) (2 pts) Go to Step 3 to instantiate and deploy the SimpleNN model on GPUs for efficient training. How can you verify that your model is indeed deployed on GPU? (Hint: use `nvidia-smi` command in the terminal)



(a) GPU usage changes while the model runs.

(b) GPU usage before model starts.

Figure 4: Demonstration that the model is on the GPU using the command `nvidia-smi`.

The changes in the GPU's memory usage and GPU usage in the figure above shows that the model is indeed deployed on the GPU. The model is deployed using `model=model.to(device)` (see step 3 of simplenn-cifar10.ipynb for more details).

(e) (2pts) Loss functions are used to encode the learning objective. Now, we need to define this problem's loss function as well as the optimizer which will update our model's parameters to minimize the loss. In Step 4, please fill out the loss function and optimizer part.
My loss function is defined as `criterion = nn.CrossEntropyLoss()`; my optimiser is defined as `optimizer = torch.optim.SGD(model.parameters(), lr=INITIAL_LR, momentum=MOMENTUM, weight_decay=REG)`. See step 4 of simplenn-cifar10.ipynb for more details.

(f) (2 pts) Please go to Step 5 to set up the training process of SimpleNN on the CIFAR-10 dataset. Follow the detailed instructions in Step 5 for guidance.
Please see step 5 of simplenn-cifar10.ipynb.

(g) (3pts) You can start training now with the provided hyperparameter setting. What is the initial loss value before you conduct any training step? How is it related to the number of classes in CIFAR-10? What can you observe from training accuracy and validation accuracy? Do you notice any problems with

the current training pipeline?

```
==> Training starts!
================================================
Epoch 0:
Training loss: 2.3048, Training accuracy: 0.1142
Validation loss: 2.3062, Validation accuracy: 0.1044
```

Figure 5: Initial loss and accuracy before training (I commented `optimizer.step()`).

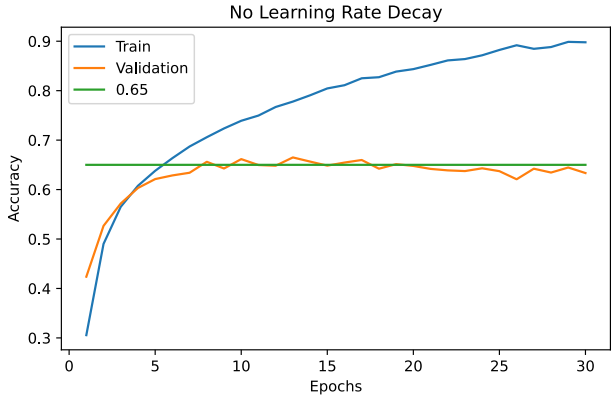
What is the initial loss value before you conduct any training step?
The initial loss value is around $\boxed{2.3}$ for both the training and validation set.

How is it related to the number of classes in CIFAR-10?
The theoretical average cross-entropy loss for a model that gives completely random guesses (assuming that model makes random guesses before it is trained) is $\boxed{l_{\text{CE}} = \ln N \approx 2.3026}$ where $N = 10$ is the number of classes in this problem (see slide 39, lecture 5). This makes sense because $s_j \approx 0.1$ for each class and this is true for all predictions. For each prediction, the loss is approximately $-1 \cdot \ln(1/10) = \ln(10)$, so for D inputs the average loss is also $\ln(10) = \ln(N)$. The accuracy before training is around $1/10$, which is $1 \div$ (number of classes in CIFAR-10).

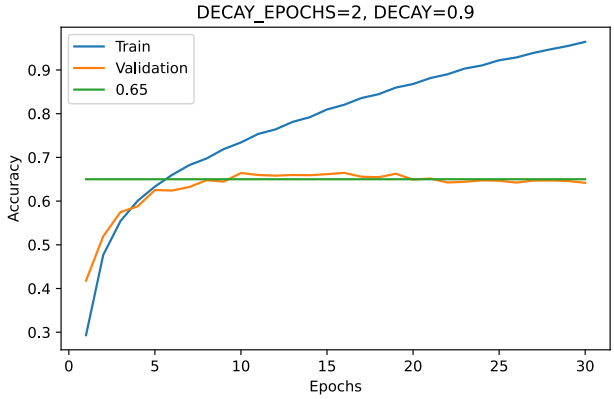
What can you observe from training accuracy and validation accuracy?
As time goes on, the **training accuracy increases steadily** with time. Initially, the validation accuracy also increases, but the **validation accuracy plateaus** at around epoch 10 and starts to slowly decrease.

Do you notice any problems with the current training pipeline?
A problem that I noticed is that the model is over-fitting the training set as indicated by the steadily increasing training accuracy and plateauing validation accuracy (see Figure 6). Data augmentation will help solve this problem. **Another problem** is that we haven't tuned any of the hyper-parameters (e.g., learning rate, L2-norm regularisation strength) and tuning them in later sections will also help improve the training/validation accuracy.

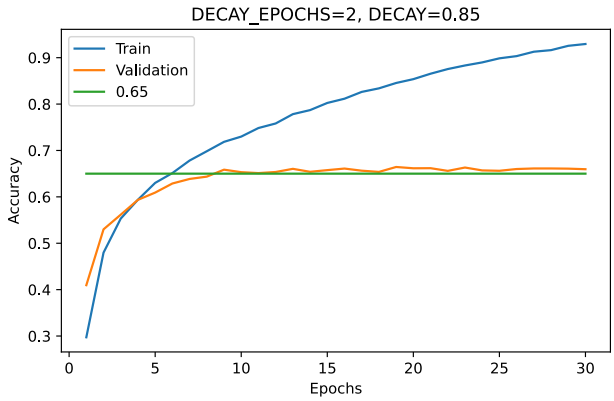
(h) (Bonus, 4 pts) Currently, we do not decay the learning rate during the training. Try to decay the learning rate (you may play with the `DECAY_EPOCHS` and `DECAY` hyperparameters in Step 5). What can you observe compared with no learning rate decay?



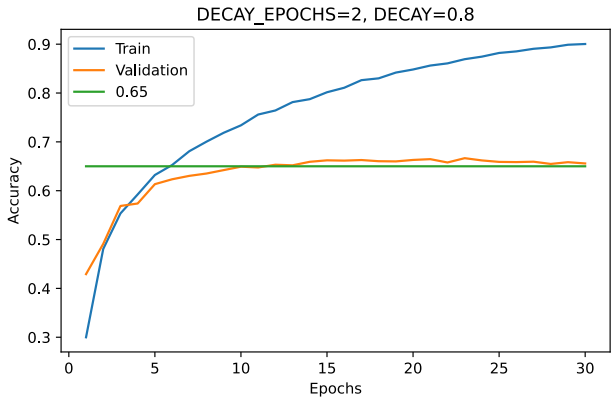
(a) Loss vs Epochs, no learning rate decay.



(b) Loss vs Epochs with learning rate decay (0.9).



(c) Loss vs Epochs with learning rate decay (0.85).



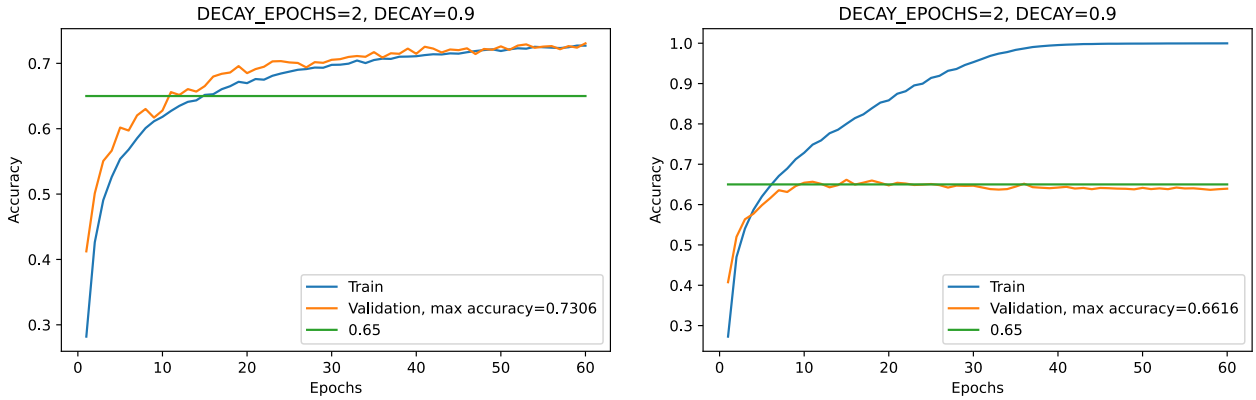
(d) Loss vs Epochs with learning rate decay (0.8).

Figure 6: Accuracy vs Epochs for different learning rate decay (learning rate is 0.01).

In Figure 6 we see that as the value for the parameter "learning rate decay" decreases from 1 to 0.8, the accuracy for the validation set decreases less towards later epochs. In other words, higher decay rates cause the model to not become worse as time goes on. For example, the validation accuracy in Figure 6a dips below 0.65 and zig-zags around 0.65 at later ecpohs, while the validation accuracy in Figure 6d stays at above 0.65 and doesn't change much from epoch to epoch after epoch 15. This is because the lower learning rate at later epochs where the learning rate decays allows the model to stay around the local minimum, whereas the relatively high learning rate for a model that doesn't have decaying learning rate causes the model to overshoot the minimum each epoch at later epochs.

3 Lab (2): Improving the training pipeline (35+6 pts)

(a) (6 pts) Data augmentation techniques help combat overfitting. A typical strategy for CIFAR classification is to combine 1) random cropping with a padding of 4 and 2) random flipping. Train a model with such augmentation. How is the validation accuracy compared with the one without augmentation?



(a) With data augmentation.

(b) Without data augmentation.

Figure 7: Accuracy vs Epochs (60 epochs) with and without data augmentation (learning rate is 0.01).

Figure 7 compares the model's accuracy. The only difference between the two models is whether data augmentation was used. The model using augmented training data (Figure 7a) has much higher validation accuracy than the model that doesn't use an augmented training set (Figure 7b). Note that the validation set isn't augmented. The relevant data-augmentation related transformations used are: `transforms.RandomCrop(size=(32, 32), padding=4, pad_if_needed=False, padding_mode='edge')`, `transforms.RandomHorizontalFlip(p=0.5)`.
Another interesting trend is that the validation accuracy more closely follows the model that uses data augmentation, which makes the model more robust and less susceptible to over-fitting by generating a different dataset for every epoch (since the transformations are random, the images that are flipped/cropped in one epoch may not necessarily be flipped/cropped in another epoch). Since the model sees somewhat different training data each epoch, the model can't "learn by heart" the features in the training set and the training step takes longer. Please see step 1 of simplenn-cifar10-dev.ipynb for more details.

(b.1) (5 pts) Add a batch normalization (BN) layer after each convolution layer. Compared with no BN layers, how does the best validation accuracy change? The model that does not use BN has maximum

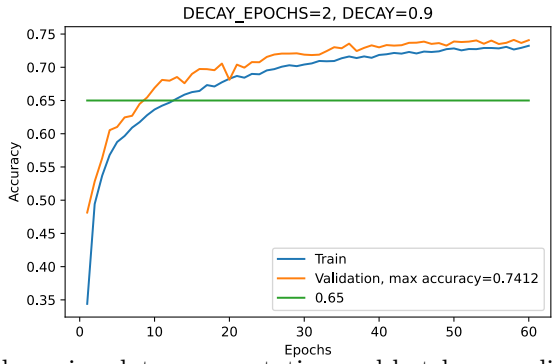
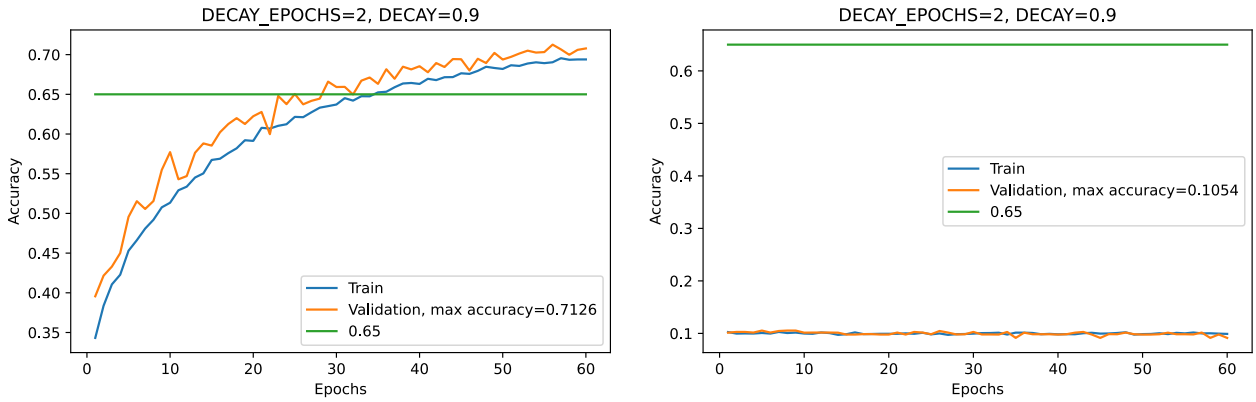


Figure 8: Accuracy vs Epochs, using data augmentation and batch normalisation (learning rate is 0.01).

validation accuracy **0.7306** (Figure 7a), while the maximum validation accuracy of the model that uses BN is slightly **higher**, at **0.7412** (Figure 8). See step 0 of simplenn-cifar10-dev.ipynb. I added `self.conv1_bn = nn.BatchNorm2d(num_features=8)` and `self.conv2_bn = nn.BatchNorm2d(num_features=16)` as member variables of the SimpleNN class and used it in the forward function (e.g. for the first convolution layer: `out = F.relu(self.conv1_bn(self.conv1(x)))`).

(b.2) (5 pts) Use empirical results to show that batch normalization allows a larger learning rate.



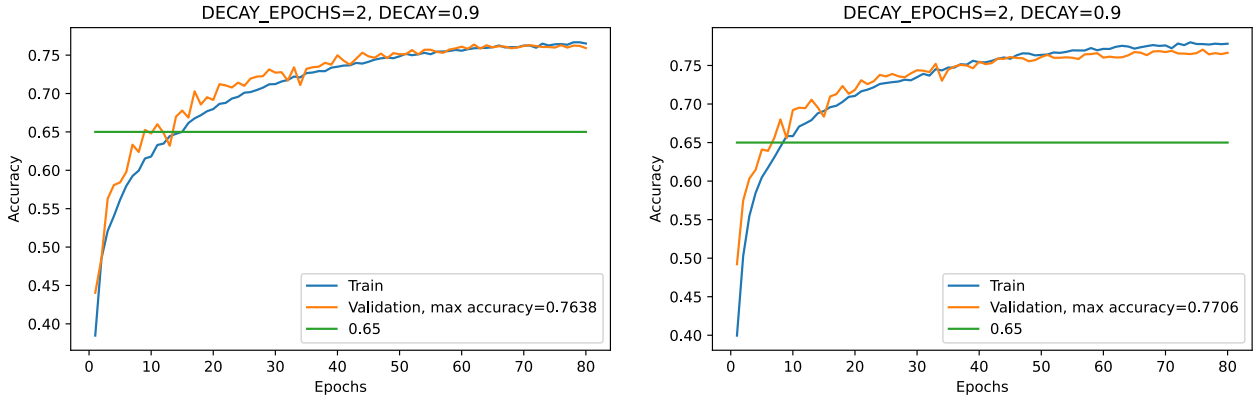
(a) With BN.

(b) Without BN.

Figure 9: Accuracy vs Epochs (60 epochs) with and without BN (**learning rate** is **0.25**, with data augmentation).

Batch normalisation does indeed allow a higher learning rate. The accuracy vs epochs plot in Figure 9a uses a model that implements BN after convolution layers, while the model in Figure 9b does not implement BN. Both models use a learning rate of 0.25. We can see that the model that implements BN is able to learn the important features of the data, while the model that doesn't implement BN doesn't learn anything at all. Even though the maximum validation accuracy (**0.7126**) at $lr = 0.25$ (Figure 9a) is slightly lower than the max validation accuracy (**0.7412**) at $lr = 0.01$ (Figure 8), we see from the comparison in Figure 9 that a model that uses BN allows a larger learning rate.

(b.3) (5 pts) Implement Swish activation on you own, and replace all of the ReLU activation functions in SimpleNN to Swish. Train the model with BN layers and a learning rate of 0.1. Does Swish outperform ReLU?



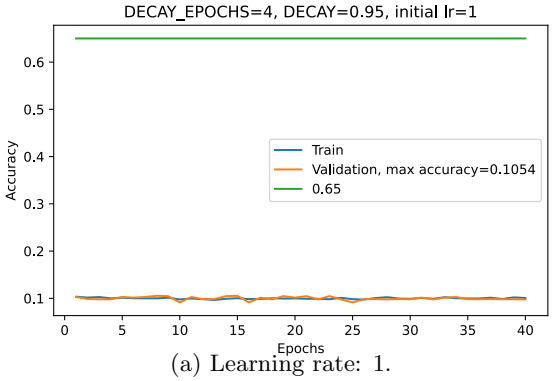
(a) With ReLU.

(b) With swish.

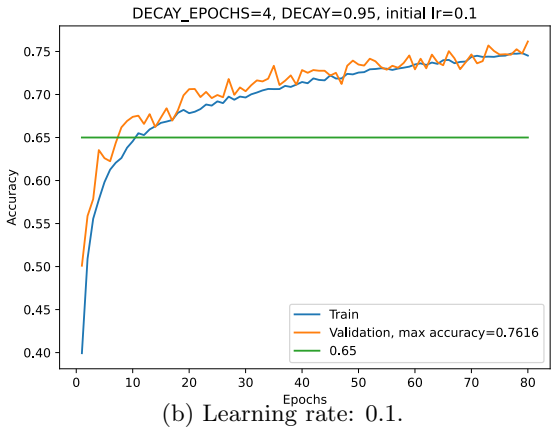
Figure 10: Accuracy vs Epochs (80 epochs), with ReLU or Swish activation (learning rate is 0.1, with data augmentation and BN).

Swish does out-perform ReLU after 80 epochs. The maximum validation accuracy with ReLU is **0.7638** (Figure 10a) while the maximum validation accuracy with Swish is **0.7706**. Note: the validation accuracy for ReLU activation (Figure 10a) here is higher than that on the previous page (Figure 8) because I'm using more epochs and a different learning rate. See step 0 of simplenn-cifar10-dev.ipynb for implementation of Swish. For example, for the first convolution layer I substituted `F.relu(self.conv1_bn(self.conv1(x)))` with `my_swish(self.conv1_bn(self.conv1(x)))`.

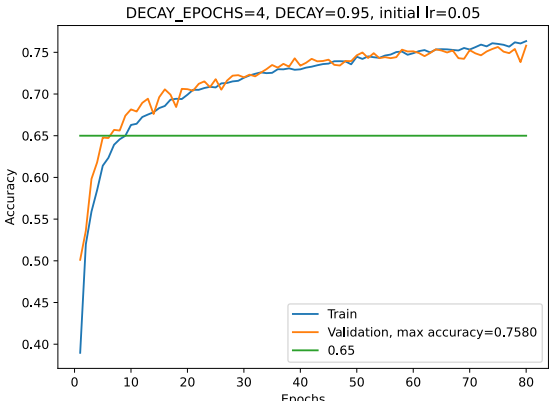
(c.1) (7 pts) Apply different learning rate values: $1.0, 0.1, 0.05, 0.01, 0.005, 0.001$, to see how the learning rate affects the model performance, and report results for each. Is a large learning rate beneficial for model training? If not, what can you conclude from the choice of learning rate?



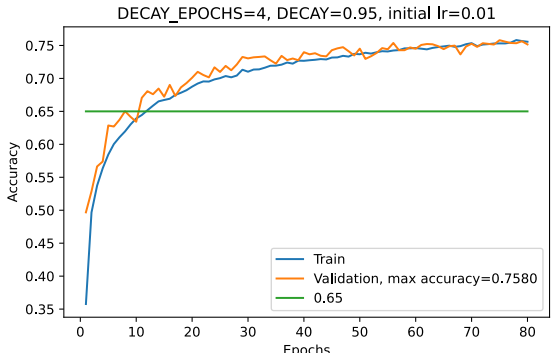
(a) Learning rate: 1.



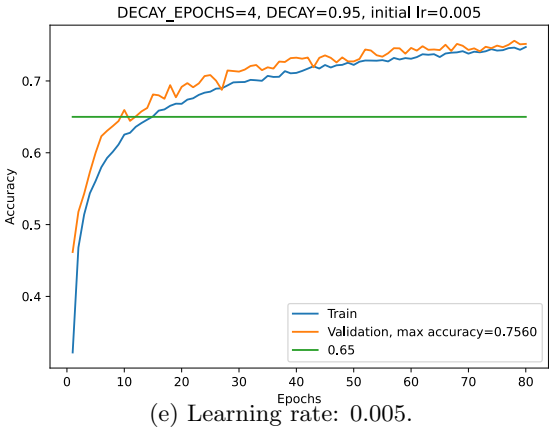
(b) Learning rate: 0.1.



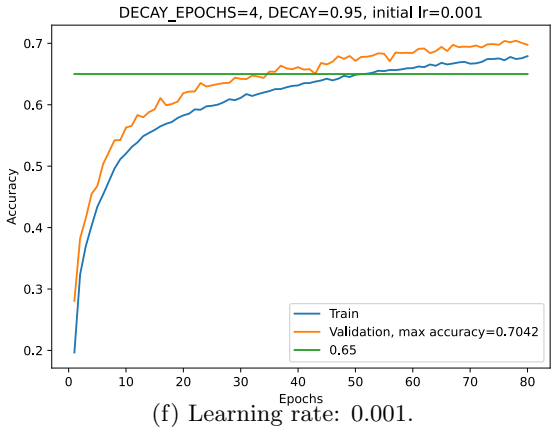
(c) Learning rate: 0.05.



(d) Learning rate: 0.01.



(e) Learning rate: 0.005.

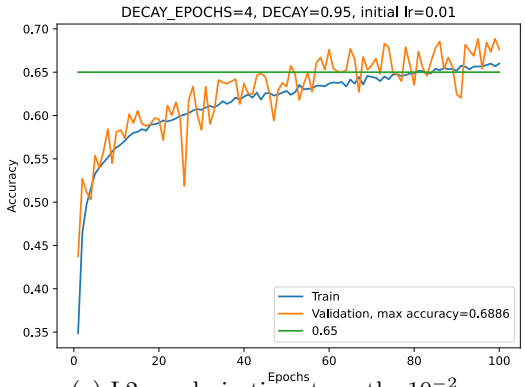


(f) Learning rate: 0.001.

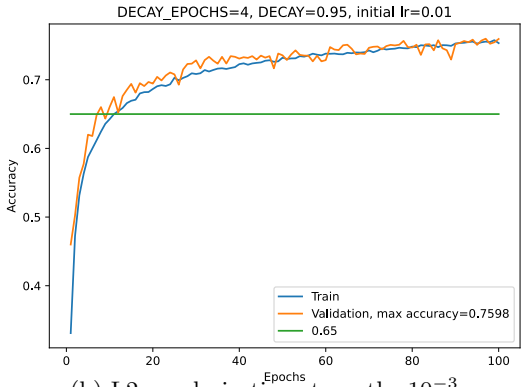
Figure 11: Accuracy vs Epochs for different learning rate values (with Swish activation, data augmentation and BN).

As learning rate increases, the maximum validation accuracy increases slightly until the learning rate is 0.1, after which the validation accuracy decreases sharply. A learning rate that is **too large** is **not beneficial** for model training (see Figure 11a). **Conclusion**: the learning rate cannot be too large (≈ 1) or too small (≈ 0.001); in the former case, the model always steps over the global minimum; in the latter case, the model takes too long to get to the global minimum (or gets stuck on a local minimum).

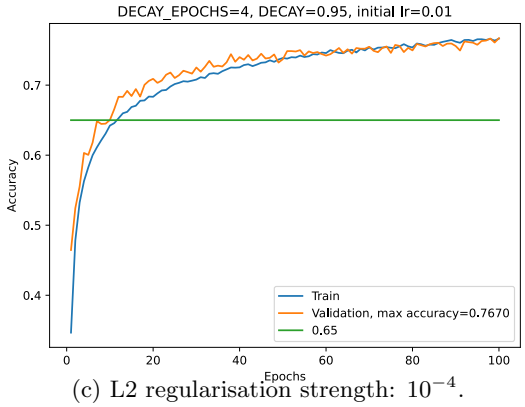
(c.2) (7 pts) Use different L2 regularisation strengths of $10^{-2}, 10^{-3}, 10^{-4}, 10^{-5}$, and 0 to see how the L2 regularization strength affects the model performance. In this problem use a learning rate of 0.01. Report the results for each regularization strength value along with comments on the importance of this hyperparameter.



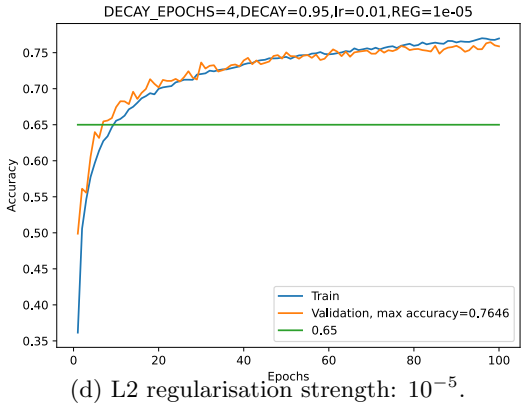
(a) L2 regularisation strength: 10^{-2}.



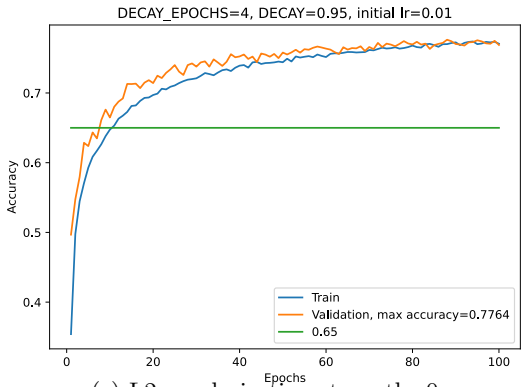
(b) L2 regularisation strength: 10^{-3}.



(c) L2 regularisation strength: 10^{-4}.



(d) L2 regularisation strength: 10^{-5}.

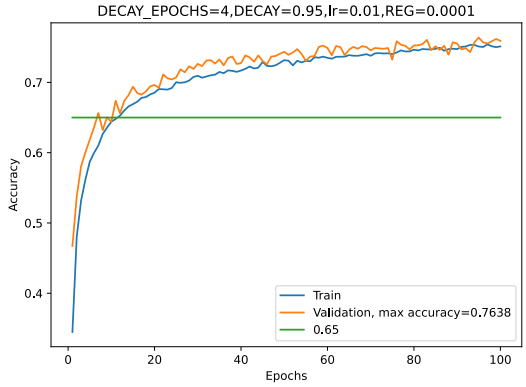


(e) L2 regularisation strength: 0.

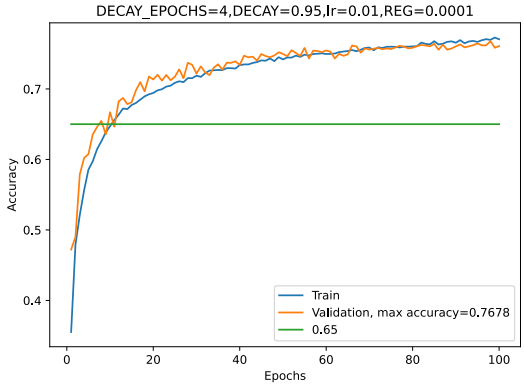
Figure 12: Accuracy vs Epochs for different regularization strength (with Swish activation, data augmentation and BN, lr = 0.01).

No regularisation seems to be the best choice for this model. Randomised on-the-fly data augmentation has already reduced the model's tendency to over-fit on the training data, so additional regularisation may not be necessary at all. The validation accuracy values for REG = $\{10^{-3}, 10^{-4}, 10^{-5}\}$ are largely the same. At REG = 10^{-2}, the best validation accuracy decreases significantly and causes the validation accuracy curve to zig-zag a lot more. When the regularisation strength is too high, the model begins to ignore the training data since weight parameters are being driven to zero. When the regularisation is too low, the model may over-fit the training data. However, for this model, the effects of on-the-fly data augmentation significantly reduced over-fitting as the training data is different for each epoch, which reduces the effects of L2 regularisation.

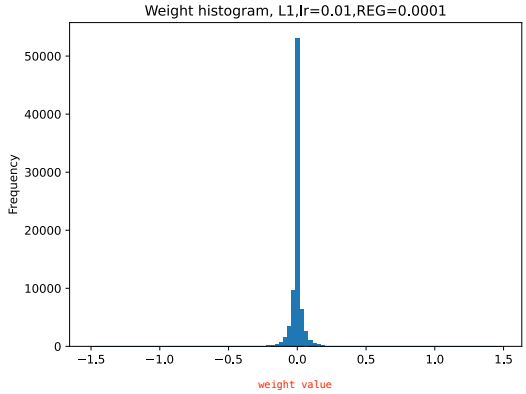
(c.3) (Bonus, 6 pts) Switch the regularisation penalty from L2 penalty to L1 penalty. This means you may not use the weight_decay parameter in PyTorch built-in optimisers, as it does not support L1 regularization. Instead, you need to add L1 penalty as a part of the loss function. Compare the distribution of weight parameters after L1/L2 regularization. Describe your observations.



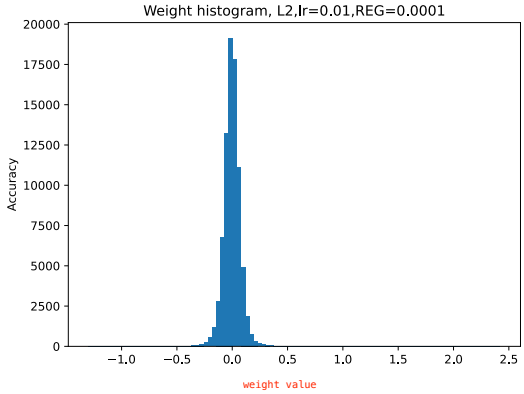
(a) L1 regularisation.



(b) L2 regularisation.



(c) L1 regularisation weight distribution.

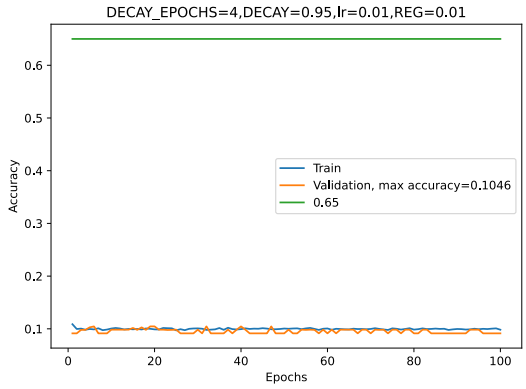


(d) L2 regularisation weight distribution.

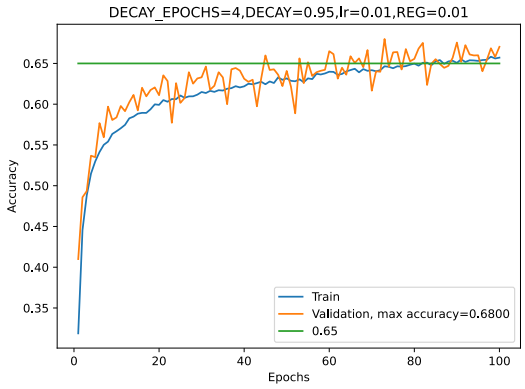
Figure 13: Effects of L1/L2 regularisation (with Swish activation, data augmentation and BN, lr = 0.01, REG = 10^{-4}).

When using low regularisation strength, the model performs similarly well with either L1 or L2 regularisation (compare Figures 13a and 13b). However, we can clearly see that the choice of regularisation method affects the distribution of weights. We can see that the weights are sparser when using L1 regularisation (i.e., weight values tend to be zero more often), while the weights have larger spread when using L2 regularisation. Note that the y-axis in Figure 13c has a different scale than the y-axis in Figure 13d; this means that there are much more weights that are zero when L1-norm regularisation is implemented compared to L2-norm. This is expected because L1 regularisation results in sparser weights in general (slide 7, Lecture 6).
Please see step 4 of simplenn-cifar10-dev.ipynb for more details on setting up the optimiser with L1-norm regularisation. Please see step 5 of simplenn-cifar10-dev.ipynb for the implementation of L1-norm regularisation. When implementing L1-norm regularisation, I computed the sum of element-wise absolute value of flattened copies of all the weights, multiplied this by `REG`, and added them to the loss all before `optimizer.zero_grad()`, `loss.backward()` and `optimizer.step()`.

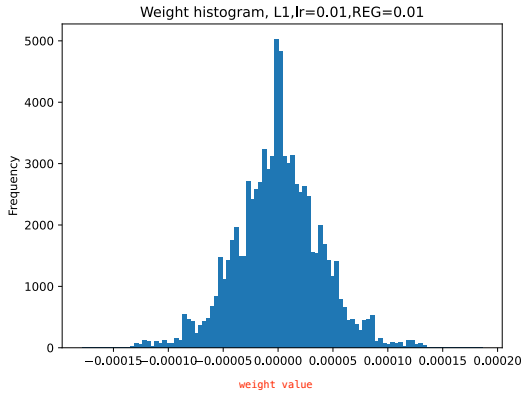
(c.3) (Bonus, 6 pts) continued



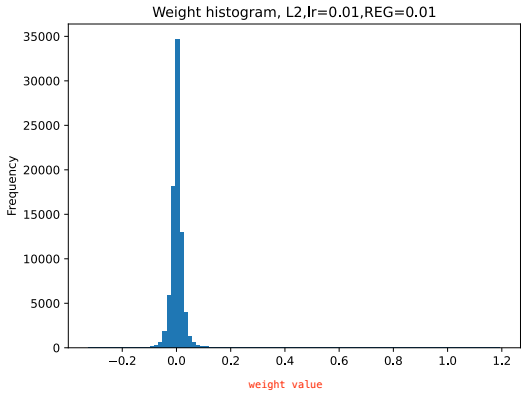
(a) L1 regularisation.



(b) L2 regularisation.



(c) L1 regularisation weight distribution.



(d) L2 regularisation weight distribution.

Figure 14: Effects of L1/L2 regularisation (with Swish activation, data augmentation and BN, lr = 0.01, REG = 10^{-2}).

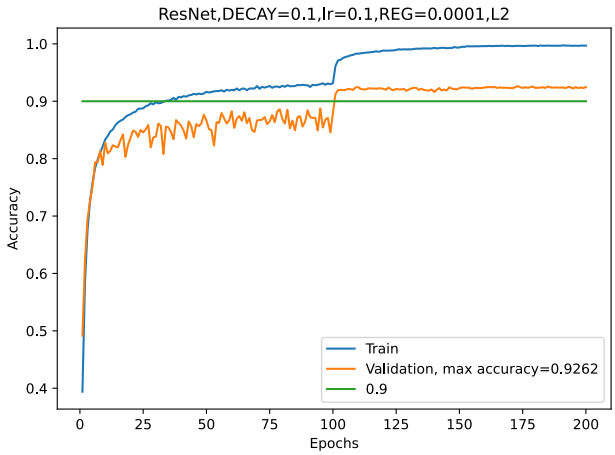
However, using L1 regularisation with a high regularisation strength is problematic. As seen in Figure 14a, the SimpleNN model fails to learn the patterns of the training data when REG = 10^{-2}, while the model is able to be train when using L2 (Figure 14b). The reason behind this is shown in Figure 14c, where the model using L1 has very small weights. This means that the model isn't learning the features of the data at all and is just outputting mostly zeros.

4 Lab (3): Advanced CNN architectures (20 pts)

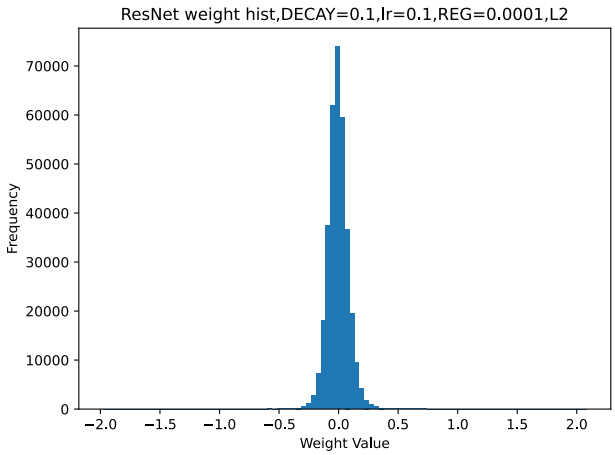
(a) (8 pts) Implement the ResNet-20 architecture by following Section 4.2 of the ResNet paper (link). See Jupyter Notebook for everything training process. The design is shown below.

```
class Residual(nn.Module):
    def __init__(self, in_channels, out_channels, stride):
        """
        :param in_channels: number of channels for the input
        :param out_channels: number of channels for the output
        :param stride: stride of first conv2d block
        :param downsample: downsampling function (option A)
        """
        # conv1: downsamples feature map when stride != 1 (padding = 1 to ensure correct shape),
        # batch norm then ReLU
        super(Residual, self).__init__()
        self.conv1 = nn.Sequential(nn.Conv2d(in_channels, out_channels, 3, stride=stride, padding=1),
                                   nn.BatchNorm2d(num_features=out_channels, eps=1e-05, momentum=0.1),
                                   nn.ReLU())
        # conv2: doesn't downsample feature map since already performed in self.conv1, batch norm,
        # padding=1 to ensure correct output shape (conv2 outputs same shape as conv1's output)
        self.conv2 = nn.Sequential(nn.Conv2d(out_channels, out_channels, 3, stride=1, padding=1),
                                   nn.BatchNorm2d(num_features=out_channels, eps=1e-05, momentum=0.1))
        self.relu2 = nn.ReLU()  # applied after adding x in the forward function

        # in case we need these
        self.in_channels = in_channels
        self.out_channels = out_channels

    def forward(self, x):
        out = self.conv1(x)
        out = self.conv2(out)
        # if number of channels for input/output mismatch, must downsample
        # if x's num_channels doesn't match out's desired n_channels:
        if x.size(dim=1) != self.out_channels:
            x = x[:, :, ::2, ::2]  # downsample feature map by a factor of 2
            # following option (A), concatenate a zeros to increase n_channels
            # (bottom-left page776 of IEEE version).
            zzzzz = torch.zeros(x.size(), device=x.device)  # get zeros of same size as x
            x_new = torch.cat((x, zzzzz), 1)  # concatenate channels
        else:
            x_new = x

        assert out.size() == x_new.size(), "out.size=={:s} != x.size=={:s}".format(str(list(out.size())),
                                                                                   str(list(x.size())))
        return self.relu2(out + x_new)


class ResNet(nn.Module):
    def __init__(self):
        super(ResNet, self).__init__()
        # implementing ResNet-20, so n=3
        # first conv to get 16-chan output, then (6x conv(in_channel=32), 6x conv(in_channel=16),
        # 6x conv(in_channel=8)), then average pooling, then FC, finally softmax
        # maintain feature map size, increase n_channels to 16 after 1st convolution
        self.conv_initial = nn.Sequential(nn.Conv2d(3, 16, 3, stride=1, padding=1),
                                          nn.BatchNorm2d(num_features=16, eps=1e-05, momentum=0.1),
                                          nn.ReLU())  # output 32*32 feature map

        # first six conv layers (ReLU, batchNorm are applied within Residual)
        self.conv32_0 = Residual(in_channels=16, out_channels=16, stride=1)  # output 32*32 feature map
        self.conv32_2 = Residual(in_channels=16, out_channels=16, stride=1)  # output 32*32 feature map
        self.conv32_4 = Residual(in_channels=16, out_channels=16, stride=1)  # output 32*32 feature map

        # feature map is 16*16 after conv16_0
        # next six conv layers, (ReLU, batchNorm are applied within Residual)
        self.conv16_0 = Residual(in_channels=16, out_channels=32, stride=2)  # output 16*16 feature map
        self.conv16_2 = Residual(in_channels=32, out_channels=32, stride=1)  # output 16*16 feature map
        self.conv16_4 = Residual(in_channels=32, out_channels=32, stride=1)  # output 16*16 feature map

        # feature map decreases to 8*8 after conv8_0
        # last six conv layers, (ReLU, batchNorm are applied within Residual)
        self.conv8_0 = Residual(in_channels=32, out_channels=64, stride=2)  # output 8*8 feature map
        self.conv8_2 = Residual(in_channels=64, out_channels=64, stride=1)  # output 8*8 feature map
        self.conv8_4 = Residual(in_channels=64, out_channels=64, stride=1)  # output 8*8 feature map
        # output after self.conv8_4 should be 64*8*8

        # global average pooling (64*(8*8) -> 64*(1))
        self.glbal_avg_pool = nn.AvgPool2d(kernel_size=8)

        # fully-conneted layer, in_channels=64, out_channels=10
        self.fc = nn.Sequential(nn.Linear(64, 10))

    def forward(self, x):
        out = self.conv_initial(x)
        out = self.conv32_0(out)
        out = self.conv32_2(out)
        out = self.conv32_4(out)

        out = self.conv16_0(out)
        out = self.conv16_2(out)
        out = self.conv16_4(out)

        out = self.conv8_0(out)
        out = self.conv8_2(out)
        out = self.conv8_4(out)

        out = self.glbal_avg_pool(out)
        out = out.view(out.size(0), -1)  # flatten before FC
        out = self.fc(out)  # includes SoftMax
        return out
```

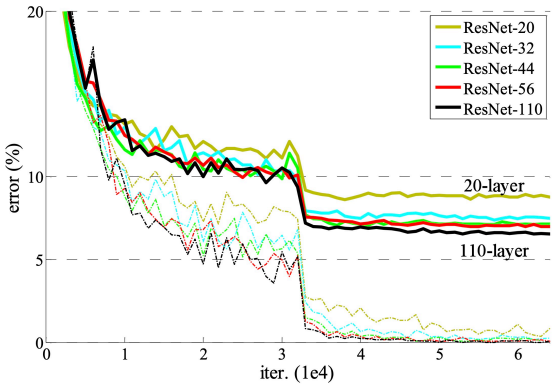
(b) (12 pts) Tune your ResNet-20 model to reach an accuracy of higher than 90% on the validation dataset.



(a) Accuracy vs Epochs.



(b) ResNet weights.



(c) ResNet results from K. He, X. Zhang, S. Ren, and J. Sun (top of Page 777 here).

Figure 15: Demonstration of working ResNet (with ReLU activation, data augmentation and BN, initial lr = 0.1, L2 REG = 10^{-4}).

Figure 15 shows that the validation accuracy of the data surpasses 90%. The learning rate started off as 0.1 and decays by a factor of 10 at epochs 100 and 150. I used ReLU activation, the SGD optimiser with momentum of 0.9, L2 regularisation (parameter `weight_decay` in the SGD optimiser) of 10^{-4}, 128 and 100 as train/validation batch sizes respectively, random cropping with 4-pixel edge padding, random horizontal flipping with flipping probability $p = 0.5$, batch normalisation, and default PyTorch weight initialisation. The model was run for 200 epochs.
Data augmentation was performed on-the-fly, meaning that no enlargement of the training set was created prior to starting the training loop; instead, the random transformations (horizontal flipping and random-cropping with 4-pixel edge padding) are applied to each mini-batch of training data right before they're fed into the model for each training iteration. One batch normalisation layer is applied after each convolution. When the feature-map size and number of channels of $f(\mathbf{x})$ and $\mathbf{x}$ do not match due to a stride-2 convolution, I down-sampled the feature maps of $\mathbf{x}$ by a factor of 2 and appended zero-valued feature maps to $\mathbf{x}$ to increase the number of channels. This is Option (A) used in the ResNet paper, which the authors specified in the bottom-left of page 776 (ResNet paper). All parameter tuning was performed with guidance from section 4.2 of the ResNet paper. Please see the Jupyter Notebook for more details. (resnet-cifar10.ipynb).
My results (Figure 15a) closely resemble those of the ResNet paper's (Figure 15c).

simplenn-cifar10

October 3, 2023

1 Training SimpleNN on CIFAR-10

In this project, you will use the SimpleNN model to perform image classification on CIFAR-10. CIFAR-10 orginally contains 60K images from 10 categories. We split it into 45K/5K/10K images to serve as train/valiation/test set. We only release the ground-truth labels of training/validation dataset to you.

1.1 Step 0: Set up the SimpleNN model

As you have practiced to implement simple neural networks in Homework 1, we just prepare the implementation for you.

[2]:
```
# import necessary dependencies
import argparse
import os, sys
import time
import datetime
from tqdm import tqdm_notebook as tqdm

import torch
import torch.nn as nn
import torch.nn.functional as F
```

[3]:
```
# define the SimpleNN mode;
class SimpleNN(nn.Module):
    def __init__(self):
        super(SimpleNN, self).__init__()
        self.conv1 = nn.Conv2d(3, 8, 5)
        self.conv2 = nn.Conv2d(8, 16, 3)
        self.fc1   = nn.Linear(16*6*6, 120)
        self.fc2   = nn.Linear(120, 84)
        self.fc3   = nn.Linear(84, 10)

    def forward(self, x):
        out = F.relu(self.conv1(x))
        out = F.max_pool2d(out, 2)
        out = F.relu(self.conv2(out))
        out = F.max_pool2d(out, 2)
```

```
        out = out.view(out.size(0), -1)
        out = F.relu(self.fc1(out))
        out = F.relu(self.fc2(out))
        out = self.fc3(out)
        return out
```

1.1.1 Question (a)

Here is a sanity check to verify the implementation of SimpleNN. You need to: 1. Write down your code. 2. **In the PDF report**, give a brief description on how the code helps you know that SimpleNN is implemented correctly.

[4]:
```
##############################################
# your code here
# sanity check for the correctness of SimpleNN
test_in = torch.rand((1, 3, 32, 32))
test_model = SimpleNN()
test_out = test_model.forward(test_in)
print(test_out.shape)
##############################################
```

```
torch.Size([1, 10])
```

1.2 Step 1: Set up preprocessing functions

Preprocessing is very important as discussed in the lecture. You will need to write preprocessing functions with the help of *torchvision.transforms* in this step. You can find helpful tutorial/API at here.

1.2.1 Question (b)

For the question, you need to: 1. Complete the preprocessing code below. 2. **In the PDF report**, briefly describe what preprocessing operations you used and what are the purposes of them.

Hint: 1. Only two operations are necessary to complete the basic preprocessing here. 2. The raw input read from the dataset will be PIL images. 3. Data augmentation operations are not mendatory, but feel free to incorporate them if you want. 4. Reference value for mean/std of CIFAR-10 images (assuming the pixel values are within [0,1]): mean (RGB-format): (0.4914, 0.4822, 0.4465), std (RGB-format): (0.2023, 0.1994, 0.2010)

[5]:
```
# useful libraries
import torchvision
import torchvision.transforms as transforms

##############################################
# your code here
# specify preprocessing function ToTensor (PIL to tensor) and Normalise (mean␣
 ↪becomes 0, stdev becomes 1)
# please see pdf for more details
```

```
transform_train =  transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.4914, 0.4822, 0.4465), (0.2023, 0.1994, 0.2010))
    ])

transform_val = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.4914, 0.4822, 0.4465), (0.2023, 0.1994, 0.2010))
    ])
################################################
```

1.3 Step 2: Set up dataset and dataloader

1.3.1 Question (c)

Set up the train/val datasets and dataloders that are to be used during the training. Check out the official API for more information about **torch.utils.data.DataLoader**.

Here, you need to: 1. Complete the code below.

[6]:
```
# do NOT change these
from tools.dataset import CIFAR10
from torch.utils.data import DataLoader

# a few arguments, do NOT change these
DATA_ROOT = "./data"
TRAIN_BATCH_SIZE = 128
VAL_BATCH_SIZE = 100

################################################
# your code here
# construct dataset
train_set = CIFAR10(
    root=DATA_ROOT,
    mode='train',
    download=True,
    transform=transform_train    # your code
)
val_set = CIFAR10(
    root=DATA_ROOT,
    mode='val',
    download=True,
    transform=transform_val    # your code
)

# construct dataloader
train_loader = DataLoader(
```

```
    train_set,
    batch_size=TRAIN_BATCH_SIZE,  # your code
    shuffle=True,      # your code
    num_workers=4
)
val_loader = DataLoader(
    val_set,
    batch_size=VAL_BATCH_SIZE,  # your code
    shuffle=False,      # your code
    num_workers=4
)
################################################
```

```
Using downloaded and verified file: ./data/cifar10_trainval_F22.zip
Extracting ./data/cifar10_trainval_F22.zip to ./data
Files already downloaded and verified
Using downloaded and verified file: ./data/cifar10_trainval_F22.zip
Extracting ./data/cifar10_trainval_F22.zip to ./data
Files already downloaded and verified
```

1.4 Step 3: Instantiate your SimpleNN model and deploy it to GPU devices.

1.4.1 Question (d)

You may want to deploy your model to GPU device for efficient training. Please assign your model to GPU if possible. If you are training on a machine without GPUs, please deploy your model to CPUs.

Here, you need to: 1. Complete the code below. 2. **In the PDF report**, briefly describe how you verify that your model is indeed deployed on GPU. (Hint: check `nvidia-smi`.)

[7]:
```
# specify the device for computation
################################################
# your code here

device = 'cuda' if torch.cuda.is_available() else 'cpu'
print(device)
# Construct our model by instantiating the class defined above
model = SimpleNN()
# Copy to CUDA device. This is very important.
model = model.to(device)
################################################
```

```
cuda
```

1.5 Step 4: Set up the loss function and optimizer

Loss function/objective function is used to provide "feedback" for the neural networks. Typically, we use multi-class cross-entropy as the loss function for classification models. As for the optimizer,

we will use SGD with momentum.

1.5.1 Question (e)

Here, you need to: 1. Set up the cross-entropy loss as the criterion. (Hint: there are implemented functions in **torch.nn**) 2. Specify a SGD optimizer with momentum. (Hint: there are implemented functions in **torch.optim**)

[8]:
```
import torch.nn as nn
import torch.optim as optim

# hyperparameters, do NOT change right now
# initial learning rate
INITIAL_LR = 0.01

# momentum for optimizer
MOMENTUM = 0.9

# L2 regularization strength
REG = 1e-4

################################################
# your code here
# create loss function
criterion = nn.CrossEntropyLoss()

# Add optimizer
optimizer = torch.optim.SGD(model.parameters(), lr=INITIAL_LR,␣
 ↪momentum=MOMENTUM, weight_decay=REG)
################################################
```

1.6 Step 5: Start the training process.

1.6.1 Question (f)/(g)

Congratulations! You have completed all of the previous steps and it is time to train our neural network.

Here you need to: 1. Complete the training codes. 2. Actually perform the training.

Hint: Training a neural network usually repeats the following 4 steps:

i) Get a batch of data from the dataloader and copy it to your device (GPU).

ii) Do a forward pass to get the outputs from the neural network and compute the loss. Be careful about your inputs to the loss function. Are the inputs required to be the logits or softmax probabilities?)

iii) Do a backward pass (back-propagation) to compute gradients of all weights with respect to the loss.

iiii) Update the model weights with the optimizer.

You will also need to compute the accuracy of training/validation samples to track your model's performance over each epoch (the accuracy should be increasing as you train for more and more epochs).

[9]:
```
# some hyperparameters
# total number of training epochs
EPOCHS = 60

DECAY_EPOCHS = 2
DECAY = 0.9

# the folder where the trained model is saved
CHECKPOINT_FOLDER = "./saved_model"

# start the training/validation process
# the process should take about 5 minutes on a GTX 1070-Ti
# if the code is written efficiently.
best_val_acc = 0
current_learning_rate = INITIAL_LR




##### my addition
accuracy_arr_train = []
accuracy_arr_val = []
##### end of my addition




print("==> Training starts!")
print("="*50)
for i in range(0, EPOCHS):
    # handle the learning rate scheduler.
    if i % DECAY_EPOCHS == 0 and i != 0:
        current_learning_rate = current_learning_rate * DECAY
        for param_group in optimizer.param_groups:
            param_group['lr'] = current_learning_rate
        print("Current learning rate has decayed to %f" %current_learning_rate)

    ########################
    # your code here
    # switch to train mode
    model.train()
```

```
    ########################

    print("Epoch %d:" %i)
    # this help you compute the training accuracy
    total_examples = 0
    correct_examples = 0

    train_loss = 0 # track training loss if you want

    # Train the model for 1 epoch.
    for batch_idx, (inputs, targets) in enumerate(train_loader):
        # if batch_idx == 0 and i == 0:
        #     print(torch.mean(inputs.flatten()))
        ###################################
        # your code here
        # copy inputs to device
        inputs, targets = inputs.to(device), targets.to(device)

        # compute the output and loss
        y_pred = model(inputs)
        loss = criterion(y_pred, targets)
        train_loss += loss
        # zero the gradient
        optimizer.zero_grad()

        # backpropagation
        loss.backward()

        # apply gradient and update the weights
        optimizer.step()

        # count the number of correctly predicted samples in the current batch
        total_examples += len(targets.view(targets.size(0), -1))
        correct_examples += sum(torch.argmax(y_pred, dim=1, keepdim=False) ==␣
↪targets)

        ###################################

    avg_loss = train_loss / len(train_loader)
    avg_acc = correct_examples / total_examples
    print("Training loss: %.4f, Training accuracy: %.4f" %(avg_loss, avg_acc))



    ##### my addition
    accuracy_arr_train.append(avg_acc)
    ##### end of my addition
```

```
    # Validate on the validation dataset
    ########################
    # your code here
    # switch to eval mode
    model.eval()

    ########################

    # this helps you compute the validation accuracy
    total_examples = 0
    correct_examples = 0

    val_loss = 0 # again, track the validation loss if you want

    # disable gradient during validation, which can save GPU memory
    with torch.no_grad():
        for batch_idx, (inputs, targets) in enumerate(val_loader):
            ###################################
            # your code here
            # copy inputs to device
            inputs, targets = inputs.to(device), targets.to(device)

            # compute the output and loss
            y_pred = model(inputs)
            loss = criterion(y_pred, targets)
            val_loss += loss

            # count the number of correctly predicted samples in the current␣
↪batch
            total_examples += len(targets.view(targets.size(0), -1))
            correct_examples += sum(torch.argmax(y_pred, dim=1, keepdim=False)␣
↪== targets)
            ###################################

    avg_loss = val_loss / len(val_loader)
    avg_acc = correct_examples / total_examples
    print("Validation loss: %.4f, Validation accuracy: %.4f" % (avg_loss,␣
↪avg_acc))



    ##### my addition
    accuracy_arr_val.append(avg_acc)
    ##### end of my addition
```

```
    # save the model checkpoint
    if avg_acc > best_val_acc:
        best_val_acc = avg_acc
        # if not os.path.exists(CHECKPOINT_FOLDER):
        #    os.makedirs(CHECKPOINT_FOLDER)
        # print("Saving ...")
        # state = {'state_dict': model.state_dict(),
        #          'epoch': i,
        #          'lr': current_learning_rate}
        # torch.save(state, os.path.join(CHECKPOINT_FOLDER, 'simplenn.pth'))

    print('')

print("="*50)
print(f"==> Optimization finished! Best validation accuracy: {best_val_acc:.
  ↪4f}")
```

```
==> Training starts!
==================================================
Epoch 0:
Training loss: 1.9625, Training accuracy: 0.2727
Validation loss: 1.6071, Validation accuracy: 0.4076

Epoch 1:
Training loss: 1.4716, Training accuracy: 0.4700
Validation loss: 1.3839, Validation accuracy: 0.5200

Current learning rate has decayed to 0.009000
Epoch 2:
Training loss: 1.2947, Training accuracy: 0.5405
Validation loss: 1.2453, Validation accuracy: 0.5636

Epoch 3:
Training loss: 1.1725, Training accuracy: 0.5868
Validation loss: 1.1866, Validation accuracy: 0.5772

Current learning rate has decayed to 0.008100
Epoch 4:
Training loss: 1.0768, Training accuracy: 0.6200
Validation loss: 1.1469, Validation accuracy: 0.5984

Epoch 5:
Training loss: 1.0068, Training accuracy: 0.6466
Validation loss: 1.0857, Validation accuracy: 0.6162
```

```
Current learning rate has decayed to 0.007290
Epoch 6:
Training loss: 0.9371, Training accuracy: 0.6709
Validation loss: 1.0392, Validation accuracy: 0.6356

Epoch 7:
Training loss: 0.8823, Training accuracy: 0.6894
Validation loss: 1.0673, Validation accuracy: 0.6312

Current learning rate has decayed to 0.006561
Epoch 8:
Training loss: 0.8204, Training accuracy: 0.7125
Validation loss: 1.0355, Validation accuracy: 0.6460

Epoch 9:
Training loss: 0.7790, Training accuracy: 0.7285
Validation loss: 1.0057, Validation accuracy: 0.6542

Current learning rate has decayed to 0.005905
Epoch 10:
Training loss: 0.7194, Training accuracy: 0.7486
Validation loss: 1.0120, Validation accuracy: 0.6566

Epoch 11:
Training loss: 0.6883, Training accuracy: 0.7591
Validation loss: 1.0438, Validation accuracy: 0.6514

Current learning rate has decayed to 0.005314
Epoch 12:
Training loss: 0.6451, Training accuracy: 0.7768
Validation loss: 1.1184, Validation accuracy: 0.6428

Epoch 13:
Training loss: 0.6130, Training accuracy: 0.7856
Validation loss: 1.0902, Validation accuracy: 0.6482

Current learning rate has decayed to 0.004783
Epoch 14:
Training loss: 0.5675, Training accuracy: 0.8002
Validation loss: 1.0817, Validation accuracy: 0.6616

Epoch 15:
Training loss: 0.5302, Training accuracy: 0.8143
Validation loss: 1.1328, Validation accuracy: 0.6496

Current learning rate has decayed to 0.004305
Epoch 16:
```

```
Training loss: 0.4984, Training accuracy: 0.8237
Validation loss: 1.1454, Validation accuracy: 0.6544

Epoch 17:
Training loss: 0.4636, Training accuracy: 0.8387
Validation loss: 1.1782, Validation accuracy: 0.6596

Current learning rate has decayed to 0.003874
Epoch 18:
Training loss: 0.4231, Training accuracy: 0.8528
Validation loss: 1.2403, Validation accuracy: 0.6540

Epoch 19:
Training loss: 0.4029, Training accuracy: 0.8584
Validation loss: 1.2890, Validation accuracy: 0.6476

Current learning rate has decayed to 0.003487
Epoch 20:
Training loss: 0.3637, Training accuracy: 0.8747
Validation loss: 1.3256, Validation accuracy: 0.6540

Epoch 21:
Training loss: 0.3403, Training accuracy: 0.8810
Validation loss: 1.3676, Validation accuracy: 0.6522

Current learning rate has decayed to 0.003138
Epoch 22:
Training loss: 0.3051, Training accuracy: 0.8954
Validation loss: 1.4623, Validation accuracy: 0.6486

Epoch 23:
Training loss: 0.2903, Training accuracy: 0.8997
Validation loss: 1.5443, Validation accuracy: 0.6494

Current learning rate has decayed to 0.002824
Epoch 24:
Training loss: 0.2557, Training accuracy: 0.9140
Validation loss: 1.5894, Validation accuracy: 0.6510

Epoch 25:
Training loss: 0.2384, Training accuracy: 0.9192
Validation loss: 1.6426, Validation accuracy: 0.6482

Current learning rate has decayed to 0.002542
Epoch 26:
Training loss: 0.2102, Training accuracy: 0.9316
Validation loss: 1.7288, Validation accuracy: 0.6422
```

```
Epoch 27:
Training loss: 0.1948, Training accuracy: 0.9361
Validation loss: 1.7892, Validation accuracy: 0.6470

Current learning rate has decayed to 0.002288
Epoch 28:
Training loss: 0.1703, Training accuracy: 0.9460
Validation loss: 1.8600, Validation accuracy: 0.6464

Epoch 29:
Training loss: 0.1528, Training accuracy: 0.9532
Validation loss: 1.9480, Validation accuracy: 0.6468

Current learning rate has decayed to 0.002059
Epoch 30:
Training loss: 0.1333, Training accuracy: 0.9608
Validation loss: 1.9924, Validation accuracy: 0.6426

Epoch 31:
Training loss: 0.1150, Training accuracy: 0.9686
Validation loss: 2.0966, Validation accuracy: 0.6384

Current learning rate has decayed to 0.001853
Epoch 32:
Training loss: 0.0998, Training accuracy: 0.9743
Validation loss: 2.2083, Validation accuracy: 0.6372

Epoch 33:
Training loss: 0.0892, Training accuracy: 0.9781
Validation loss: 2.2950, Validation accuracy: 0.6384

Current learning rate has decayed to 0.001668
Epoch 34:
Training loss: 0.0755, Training accuracy: 0.9835
Validation loss: 2.3423, Validation accuracy: 0.6446

Epoch 35:
Training loss: 0.0656, Training accuracy: 0.9870
Validation loss: 2.3982, Validation accuracy: 0.6518

Current learning rate has decayed to 0.001501
Epoch 36:
Training loss: 0.0557, Training accuracy: 0.9905
Validation loss: 2.5021, Validation accuracy: 0.6432

Epoch 37:
Training loss: 0.0485, Training accuracy: 0.9925
Validation loss: 2.5632, Validation accuracy: 0.6418
```

```
Current learning rate has decayed to 0.001351
Epoch 38:
Training loss: 0.0418, Training accuracy: 0.9942
Validation loss: 2.6383, Validation accuracy: 0.6406

Epoch 39:
Training loss: 0.0374, Training accuracy: 0.9954
Validation loss: 2.6903, Validation accuracy: 0.6422

Current learning rate has decayed to 0.001216
Epoch 40:
Training loss: 0.0329, Training accuracy: 0.9964
Validation loss: 2.7536, Validation accuracy: 0.6440

Epoch 41:
Training loss: 0.0297, Training accuracy: 0.9971
Validation loss: 2.8155, Validation accuracy: 0.6398

Current learning rate has decayed to 0.001094
Epoch 42:
Training loss: 0.0266, Training accuracy: 0.9979
Validation loss: 2.8463, Validation accuracy: 0.6412

Epoch 43:
Training loss: 0.0245, Training accuracy: 0.9980
Validation loss: 2.9160, Validation accuracy: 0.6382

Current learning rate has decayed to 0.000985
Epoch 44:
Training loss: 0.0224, Training accuracy: 0.9985
Validation loss: 2.9329, Validation accuracy: 0.6412

Epoch 45:
Training loss: 0.0209, Training accuracy: 0.9988
Validation loss: 2.9714, Validation accuracy: 0.6406

Current learning rate has decayed to 0.000886
Epoch 46:
Training loss: 0.0193, Training accuracy: 0.9988
Validation loss: 3.0046, Validation accuracy: 0.6396

Epoch 47:
Training loss: 0.0183, Training accuracy: 0.9990
Validation loss: 3.0422, Validation accuracy: 0.6392

Current learning rate has decayed to 0.000798
Epoch 48:
```

```
Training loss: 0.0172, Training accuracy: 0.9991
Validation loss: 3.0792, Validation accuracy: 0.6378

Epoch 49:
Training loss: 0.0165, Training accuracy: 0.9991
Validation loss: 3.1001, Validation accuracy: 0.6414

Current learning rate has decayed to 0.000718
Epoch 50:
Training loss: 0.0156, Training accuracy: 0.9992
Validation loss: 3.1270, Validation accuracy: 0.6384

Epoch 51:
Training loss: 0.0150, Training accuracy: 0.9993
Validation loss: 3.1526, Validation accuracy: 0.6402

Current learning rate has decayed to 0.000646
Epoch 52:
Training loss: 0.0143, Training accuracy: 0.9994
Validation loss: 3.1702, Validation accuracy: 0.6382

Epoch 53:
Training loss: 0.0139, Training accuracy: 0.9994
Validation loss: 3.1846, Validation accuracy: 0.6422

Current learning rate has decayed to 0.000581
Epoch 54:
Training loss: 0.0133, Training accuracy: 0.9995
Validation loss: 3.2113, Validation accuracy: 0.6400

Epoch 55:
Training loss: 0.0129, Training accuracy: 0.9996
Validation loss: 3.2298, Validation accuracy: 0.6404

Current learning rate has decayed to 0.000523
Epoch 56:
Training loss: 0.0125, Training accuracy: 0.9996
Validation loss: 3.2405, Validation accuracy: 0.6386

Epoch 57:
Training loss: 0.0122, Training accuracy: 0.9996
Validation loss: 3.2621, Validation accuracy: 0.6366

Current learning rate has decayed to 0.000471
Epoch 58:
Training loss: 0.0118, Training accuracy: 0.9996
Validation loss: 3.2762, Validation accuracy: 0.6382
```

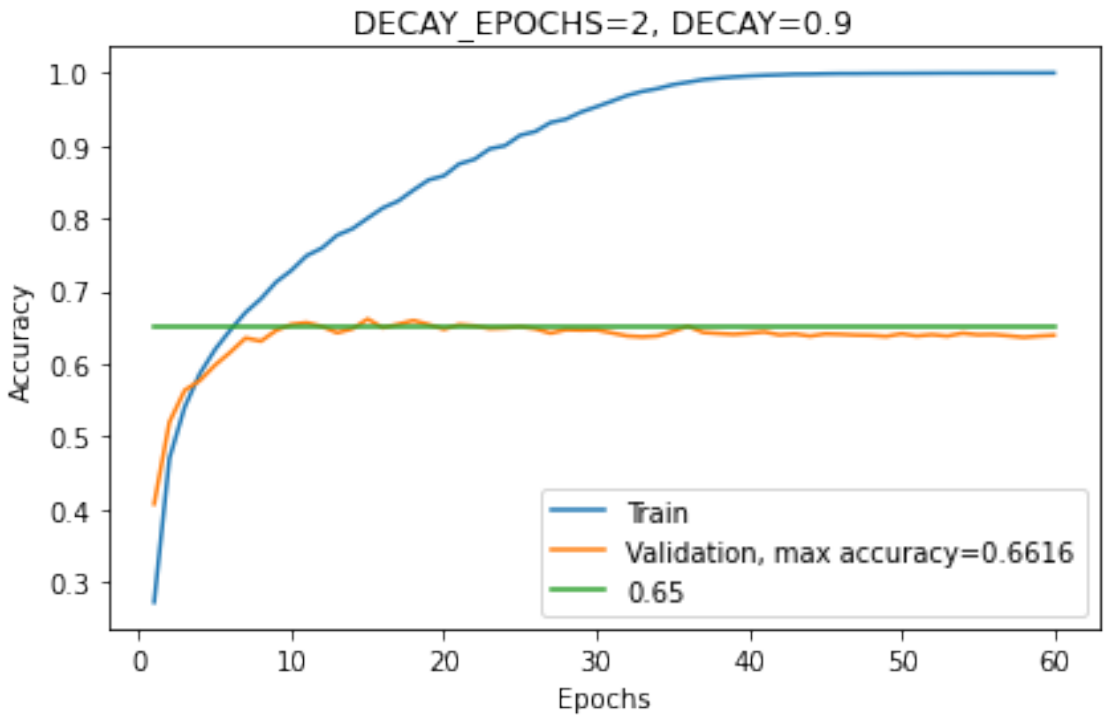
```
Epoch 59:
Training loss: 0.0115, Training accuracy: 0.9997
Validation loss: 3.2823, Validation accuracy: 0.6394

================================================
==> Optimization finished! Best validation accuracy: 0.6616
```

2 Bonus: with learning rate decay

The following code can help you adjust the learning rate during training. You need to figure out how to incorporate this code into your training loop.

```
    if i % DECAY_EPOCHS == 0 and i != 0:
        current_learning_rate = current_learning_rate * DECAY
        for param_group in optimizer.param_groups:
            param_group['lr'] = current_learning_rate
        print("Current learning rate has decayed to %f" %current_learning_rate)
```

[11]:
```
##### my addition
import matplotlib.pyplot as plt
import numpy as np

save = 'y'

fig, ax = plt.subplots(1, 1)
xx = np.linspace(1, EPOCHS, EPOCHS)

ax.plot(xx, accuracy_arr_train, label='Train')
ax.plot(xx, accuracy_arr_val, label='Validation, max accuracy={:.4f}'.
 ↪format(best_val_acc))
ax.plot(xx, np.linspace(0.65, 0.65, EPOCHS), label='0.65')
title_ = 'DECAY_EPOCHS={:d}, DECAY={:g}'.format(DECAY_EPOCHS, DECAY)
if DECAY == 1:
    title_ = 'No Learning Rate Decay'.format(DECAY_EPOCHS)
ax.set_xlabel('Epochs')
ax.set_ylabel('Accuracy')
ax.legend()
ax.set_title(title_)
fig.tight_layout()
if save == 'y':
    # plt.savefig('q1h_DECAY_EPOCHS_{:d}_DECAY_{:g}.pdf'.format(DECAY_EPOCHS,␣
 ↪DECAY), dpi=500, bbox_inches='tight')
    plt.savefig('q2a_DECAY_EPOCHS_{:d}_DECAY_{:g}_wo_aug.pdf'.
                format(DECAY_EPOCHS, DECAY), dpi=500, bbox_inches='tight')
##### end of my addition
```


DECAY_EPOCHS=2, DECAY=0.9
Accuracy
Epochs
Train
Validation, max accuracy=0.6616
0.65
1.0
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0
10
20
30
40
50
60

[ ]:

simplenn-cifar10-dev

October 3, 2023

1 Training SimpleNN on CIFAR-10

In this project, you will use the SimpleNN model to perform image classification on CIFAR-10. CIFAR-10 orginally contains 60K images from 10 categories. We split it into 45K/5K/10K images to serve as train/valiation/test set. We only release the ground-truth labels of training/validation dataset to you.

1.1 Step 0: Set up the SimpleNN model

As you have practiced to implement simple neural networks in Homework 1, we just prepare the implementation for you.

[2]:
```
# import necessary dependencies
import argparse
import os, sys
import time
import datetime
from tqdm import tqdm_notebook as tqdm

import torch
import torch.nn as nn
import torch.nn.functional as F
```

[3]:
```
def my_swish(x):
    return torch.sigmoid(x) * x
## implementing swish

# define the SimpleNN mode;
class SimpleNN(nn.Module):
    def __init__(self):
        super(SimpleNN, self).__init__()
        self.conv1 = nn.Conv2d(3, 8, 5)
        self.conv1_bn = nn.BatchNorm2d(num_features=8, eps=1e-05, momentum=0.1)
        ## adds first batch normalisation layer
        self.conv2 = nn.Conv2d(8, 16, 3)
        self.conv2_bn = nn.BatchNorm2d(num_features=16, eps=1e-05, momentum=0.1)
        ## adds second batch normalisation layer
        self.fc1   = nn.Linear(16*6*6, 120)
```

```
        self.fc2   = nn.Linear(120, 84)
        self.fc3   = nn.Linear(84, 10)

    def forward(self, x):
        # out = F.relu(self.conv1_bn(self.conv1(x)))  # relu(BN(conv))
        # out = F.relu(self.conv1(x))  # relu(conv)

        ## applies batch normalisation layer and swish
        out = my_swish(self.conv1_bn(self.conv1(x)))
        out = F.max_pool2d(out, 2)

        # out = F.relu(self.conv2_bn(self.conv2(out)))  # relu(BN(conv))
        # out = F.relu(self.conv2(out))  # relu(conv)

        ## applies batch normalisation layer and swish
        out = my_swish(self.conv2_bn(self.conv2(out)))
        out = F.max_pool2d(out, 2)

        out = out.view(out.size(0), -1)
        out = F.relu(self.fc1(out))
        out = F.relu(self.fc2(out))
        out = self.fc3(out)
        return out
```

1.1.1 Question (a)

Here is a sanity check to verify the implementation of SimpleNN. You need to: 1. Write down your code. 2. **In the PDF report**, give a brief description on how the code helps you know that SimpleNN is implemented correctly.

[4]:
```
##############################################
# your code here
# sanity check for the correctness of SimpleNN
test_in = torch.rand((1, 3, 32, 32))
test_model = SimpleNN()
test_out = test_model.forward(test_in)
print(test_out.shape)
##############################################
```

```
torch.Size([1, 10])
```

1.2 Step 1: Set up preprocessing functions

Preprocessing is very important as discussed in the lecture. You will need to write preprocessing functions with the help of *torchvision.transforms* in this step. You can find helpful tutorial/API at here.

1.2.1 Question (b)

For the question, you need to: 1. Complete the preprocessing code below. 2. **In the PDF report**, briefly describe what preprocessing operations you used and what are the purposes of them.

Hint: 1. Only two operations are necessary to complete the basic preprocessing here. 2. The raw input read from the dataset will be PIL images. 3. Data augmentation operations are not mendatory, but feel free to incorporate them if you want. 4. Reference value for mean/std of CIFAR-10 images (assuming the pixel values are within [0,1]): mean (RGB-format): (0.4914, 0.4822, 0.4465), std (RGB-format): (0.2023, 0.1994, 0.2010)

[5]:
```
# useful libraries
import torchvision
import torchvision.transforms as transforms

################################################
# your code here
# specify preprocessing function ToTensor (PIL to tensor) and Normalise (mean␣
 ↪becomes 0, stdev becomes 1)
# please see pdf for more details

transform_train =  transforms.Compose([
    transforms.ToTensor(),  ## data augmentation below for training only
    transforms.RandomCrop(size=(32, 32), padding=4, pad_if_needed=False,␣
 ↪padding_mode='edge'),
    transforms.RandomHorizontalFlip(p=0.5),
    transforms.Normalize((0.4914, 0.4822, 0.4465), (0.2023, 0.1994, 0.2010))
    ])

# validation set: no data augmentation
transform_val = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.4914, 0.4822, 0.4465), (0.2023, 0.1994, 0.2010))
    ])
################################################
```

1.3 Step 2: Set up dataset and dataloader

1.3.1 Question (c)

Set up the train/val datasets and dataloders that are to be used during the training. Check out the official API for more information about **torch.utils.data.DataLoader**.

Here, you need to: 1. Complete the code below.

[6]:
```
# do NOT change these
from tools.dataset import CIFAR10
from torch.utils.data import DataLoader
```

```
# a few arguments, do NOT change these
DATA_ROOT = "./data"
TRAIN_BATCH_SIZE = 128
VAL_BATCH_SIZE = 100

################################################
# your code here
# construct dataset
train_set = CIFAR10(
    root=DATA_ROOT,
    mode='train',
    download=True,
    transform=transform_train    # your code
)
val_set = CIFAR10(
    root=DATA_ROOT,
    mode='val',
    download=True,
    transform=transform_val    # your code
)

# construct dataloader
train_loader = DataLoader(
    train_set,
    batch_size=TRAIN_BATCH_SIZE,  # your code
    shuffle=True,     # your code
    num_workers=4
)
val_loader = DataLoader(
    val_set,
    batch_size=VAL_BATCH_SIZE,  # your code
    shuffle=True,     # your code
    num_workers=4
)
################################################
```

```
Using downloaded and verified file: ./data/cifar10_trainval_F22.zip
Extracting ./data/cifar10_trainval_F22.zip to ./data
Files already downloaded and verified
Using downloaded and verified file: ./data/cifar10_trainval_F22.zip
Extracting ./data/cifar10_trainval_F22.zip to ./data
Files already downloaded and verified
```

1.4 Step 3: Instantiate your SimpleNN model and deploy it to GPU devices.

1.4.1 Question (d)

You may want to deploy your model to GPU device for efficient training. Please assign your model to GPU if possible. If you are training on a machine without GPUs, please deploy your model to CPUs.

Here, you need to: 1. Complete the code below. 2. **In the PDF report**, briefly describe how you verify that your model is indeed deployed on GPU. (Hint: check `nvidia-smi`.)

[7]:
```
# specify the device for computation
#############################################
# your code here

device = 'cuda' if torch.cuda.is_available() else 'cpu'
print(device)
# Construct our model by instantiating the class defined above
model = SimpleNN()
# Copy to CUDA device. This is very important.
model = model.to(device)
#############################################
```

```
cuda
```

1.5 Step 4: Set up the loss function and optimizer

Loss function/objective function is used to provide "feedback" for the neural networks. Typically, we use multi-class cross-entropy as the loss function for classification models. As for the optimizer, we will use SGD with momentum.

1.5.1 Question (e)

Here, you need to: 1. Set up the cross-entropy loss as the criterion. (Hint: there are implemented functions in **torch.nn**) 2. Specify a SGD optimizer with momentum. (Hint: there are implemented functions in **torch.optim**)

[8]:
```
import torch.nn as nn
import torch.optim as optim



##### my addition
L1_or_L2 = 'L1'
##### end of my addition



# hyperparameters, do NOT change right now
# initial learning rate
```

```
INITIAL_LR = 0.01

# momentum for optimizer
MOMENTUM = 0.9

# L2 regularization strength
REG = 1e-4  # 1e-4 is orig reg strength (L2 and L1)

################################################
# your code here
# create loss function
criterion = nn.CrossEntropyLoss()

# Add optimizer
assert MOMENTUM == 0.9, "momentum isn't 0.9"

# using L2 optimisation
if L1_or_L2 == 'L2':
    optimizer = torch.optim.SGD(model.parameters(), lr=INITIAL_LR,␣
 ↪momentum=MOMENTUM, weight_decay=REG)

# using L1 optimisation, weight_decay=0 (no L2)
if L1_or_L2 == 'L1':
    optimizer = torch.optim.SGD(model.parameters(), lr=INITIAL_LR,␣
 ↪momentum=MOMENTUM, weight_decay=0)

assert L1_or_L2 == 'L1' or L1_or_L2 == 'L2', 'L1_or_L2 should be either str L1␣
 ↪or L2'
################################################
```

1.6 Step 5: Start the training process.

1.6.1 Question (f)/(g)

Congratulations! You have completed all of the previous steps and it is time to train our neural network.

Here you need to: 1. Complete the training codes. 2. Actually perform the training.

Hint: Training a neural network usually repeats the following 4 steps:

i) Get a batch of data from the dataloader and copy it to your device (GPU).

ii) Do a forward pass to get the outputs from the neural network and compute the loss. Be careful about your inputs to the loss function. Are the inputs required to be the logits or softmax probabilities?)

iii) Do a backward pass (back-propagation) to compute gradients of all weights with respect to the loss.

iiii) Update the model weights with the optimizer.

You will also need to compute the accuracy of training/validation samples to track your model's performance over each epoch (the accuracy should be increasing as you train for more and more epochs).

[9]:
```
# some hyperparameters
# total number of training epochs
EPOCHS = 80

DECAY_EPOCHS = 4
DECAY = 0.95

# the folder where the trained model is saved
CHECKPOINT_FOLDER = "./saved_model"

# start the training/validation process
# the process should take about 5 minutes on a GTX 1070-Ti
# if the code is written efficiently.
best_val_acc = 0
current_learning_rate = INITIAL_LR




##### my addition
accuracy_arr_train = []
accuracy_arr_val = []
##### end of my addition




print("==> Training starts!")
print("="*50)
for i in range(0, EPOCHS):
    # handle the learning rate scheduler.
    if i % DECAY_EPOCHS == 0 and i != 0:
        current_learning_rate = current_learning_rate * DECAY
        for param_group in optimizer.param_groups:
            param_group['lr'] = current_learning_rate
        print("Current learning rate has decayed to %f" %current_learning_rate)

    ########################
    # your code here
    # switch to train mode
    model.train()
```

```
    ########################

    print("Epoch %d:" %i)
    # this help you compute the training accuracy
    total_examples = 0
    correct_examples = 0

    train_loss = 0 # track training loss if you want

    # Train the model for 1 epoch.
    for batch_idx, (inputs, targets) in enumerate(train_loader):
        # if batch_idx == 0 and i == 0:
        #     print(torch.mean(inputs.flatten()))
        ###################################
        # your code here
        # copy inputs to device
        inputs, targets = inputs.to(device), targets.to(device)

        # compute the output and loss
        y_pred = model(inputs)
        loss = criterion(y_pred, targets)



        ## L1 normalisation
##### my addition
        if L1_or_L2 == 'L1':
            for name, param in model.named_parameters():
                if 'bias' not in name:
                    loss += REG * torch.sum(torch.abs(param))
##### end of my addition




        train_loss += loss

        # zero the gradient
        optimizer.zero_grad()

        # backpropagation
        loss.backward()

        # apply gradient and update the weights
        optimizer.step()

        # count the number of correctly predicted samples in the current batch
```

```
        total_examples += len(targets.view(targets.size(0), -1))
        correct_examples += sum(torch.argmax(y_pred, dim=1, keepdim=False) ==␣
↪targets)

        ####################################

    avg_loss = train_loss / len(train_loader)
    avg_acc = correct_examples / total_examples
    print("Training loss: %.4f, Training accuracy: %.4f" %(avg_loss, avg_acc))



##### my addition
    accuracy_arr_train.append(avg_acc)
##### end of my addition



    # Validate on the validation dataset
    ########################
    # your code here
    # switch to eval mode
    model.eval()

    ########################

    # this helps you compute the validation accuracy
    total_examples = 0
    correct_examples = 0

    val_loss = 0 # again, track the validation loss if you want

    # disable gradient during validation, which can save GPU memory
    with torch.no_grad():
        for batch_idx, (inputs, targets) in enumerate(val_loader):
            ####################################
            # your code here
            # copy inputs to device
            inputs, targets = inputs.to(device), targets.to(device)

            # compute the output and loss
            y_pred = model(inputs)
            loss = criterion(y_pred, targets)
            val_loss += loss

            # count the number of correctly predicted samples in the current␣
↪batch
```

```
            total_examples += len(targets.view(targets.size(0), -1))
            correct_examples += sum(torch.argmax(y_pred, dim=1, keepdim=False)␣
 ↪== targets)
            ####################################

    avg_loss = val_loss / len(val_loader)
    avg_acc = correct_examples / total_examples
    print("Validation loss: %.4f, Validation accuracy: %.4f" % (avg_loss,␣
 ↪avg_acc))



##### my addition
    accuracy_arr_val.append(avg_acc)
##### end of my addition



    # save the model checkpoint
    if avg_acc > best_val_acc:
        best_val_acc = avg_acc
        # if not os.path.exists(CHECKPOINT_FOLDER):
        #    os.makedirs(CHECKPOINT_FOLDER)
        # print("Saving ...")
        # state = {'state_dict': model.state_dict(),
        #          'epoch': i,
        #          'lr': current_learning_rate}
        # torch.save(state, os.path.join(CHECKPOINT_FOLDER, 'simplenn-dev.pth'))

    print('')

print("="*50)
print(f"==> Optimization finished! Best validation accuracy: {best_val_acc:.
 ↪4f}")
```

```
==> Training starts!
==================================================
Epoch 0:
Training loss: 1.9695, Training accuracy: 0.3468
Validation loss: 1.4438, Validation accuracy: 0.4830

Epoch 1:
Training loss: 1.6324, Training accuracy: 0.4778
Validation loss: 1.2835, Validation accuracy: 0.5426

Epoch 2:
Training loss: 1.4969, Training accuracy: 0.5282
```

```
Validation loss: 1.1686, Validation accuracy: 0.5896

Epoch 3:
Training loss: 1.4209, Training accuracy: 0.5537
Validation loss: 1.1410, Validation accuracy: 0.5832

Current learning rate has decayed to 0.009500
Epoch 4:
Training loss: 1.3520, Training accuracy: 0.5788
Validation loss: 1.0623, Validation accuracy: 0.6254

Epoch 5:
Training loss: 1.3077, Training accuracy: 0.5946
Validation loss: 1.0455, Validation accuracy: 0.6296

Epoch 6:
Training loss: 1.2726, Training accuracy: 0.6114
Validation loss: 0.9927, Validation accuracy: 0.6500

Epoch 7:
Training loss: 1.2434, Training accuracy: 0.6199
Validation loss: 0.9730, Validation accuracy: 0.6582

Current learning rate has decayed to 0.009025
Epoch 8:
Training loss: 1.2144, Training accuracy: 0.6328
Validation loss: 1.0359, Validation accuracy: 0.6318

Epoch 9:
Training loss: 1.2039, Training accuracy: 0.6383
Validation loss: 0.9408, Validation accuracy: 0.6682

Epoch 10:
Training loss: 1.1899, Training accuracy: 0.6428
Validation loss: 0.9123, Validation accuracy: 0.6810

Epoch 11:
Training loss: 1.1634, Training accuracy: 0.6538
Validation loss: 0.8942, Validation accuracy: 0.6824

Current learning rate has decayed to 0.008574
Epoch 12:
Training loss: 1.1556, Training accuracy: 0.6531
Validation loss: 0.9181, Validation accuracy: 0.6774

Epoch 13:
Training loss: 1.1460, Training accuracy: 0.6610
Validation loss: 0.8995, Validation accuracy: 0.6780
```

```
Epoch 14:
Training loss: 1.1447, Training accuracy: 0.6618
Validation loss: 0.9080, Validation accuracy: 0.6766

Epoch 15:
Training loss: 1.1201, Training accuracy: 0.6734
Validation loss: 0.9119, Validation accuracy: 0.6732

Current learning rate has decayed to 0.008145
Epoch 16:
Training loss: 1.1150, Training accuracy: 0.6731
Validation loss: 0.8545, Validation accuracy: 0.6982

Epoch 17:
Training loss: 1.1091, Training accuracy: 0.6742
Validation loss: 0.8717, Validation accuracy: 0.6902

Epoch 18:
Training loss: 1.1033, Training accuracy: 0.6792
Validation loss: 0.8639, Validation accuracy: 0.6904

Epoch 19:
Training loss: 1.0967, Training accuracy: 0.6817
Validation loss: 0.8311, Validation accuracy: 0.7138

Current learning rate has decayed to 0.007738
Epoch 20:
Training loss: 1.0868, Training accuracy: 0.6882
Validation loss: 0.8621, Validation accuracy: 0.6982

Epoch 21:
Training loss: 1.0844, Training accuracy: 0.6867
Validation loss: 0.8236, Validation accuracy: 0.7078

Epoch 22:
Training loss: 1.0795, Training accuracy: 0.6882
Validation loss: 0.8711, Validation accuracy: 0.6966

Epoch 23:
Training loss: 1.0714, Training accuracy: 0.6894
Validation loss: 0.8002, Validation accuracy: 0.7158

Current learning rate has decayed to 0.007351
Epoch 24:
Training loss: 1.0681, Training accuracy: 0.6917
Validation loss: 0.8195, Validation accuracy: 0.7122
```

```
Epoch 25:
Training loss: 1.0682, Training accuracy: 0.6947
Validation loss: 0.8217, Validation accuracy: 0.7118

Epoch 26:
Training loss: 1.0576, Training accuracy: 0.6974
Validation loss: 0.8289, Validation accuracy: 0.7114

Epoch 27:
Training loss: 1.0566, Training accuracy: 0.6961
Validation loss: 0.8132, Validation accuracy: 0.7136

Current learning rate has decayed to 0.006983
Epoch 28:
Training loss: 1.0500, Training accuracy: 0.6989
Validation loss: 0.7995, Validation accuracy: 0.7194

Epoch 29:
Training loss: 1.0451, Training accuracy: 0.7024
Validation loss: 0.8104, Validation accuracy: 0.7174

Epoch 30:
Training loss: 1.0433, Training accuracy: 0.7028
Validation loss: 0.8458, Validation accuracy: 0.7094

Epoch 31:
Training loss: 1.0417, Training accuracy: 0.7052
Validation loss: 0.8000, Validation accuracy: 0.7252

Current learning rate has decayed to 0.006634
Epoch 32:
Training loss: 1.0344, Training accuracy: 0.7084
Validation loss: 0.8187, Validation accuracy: 0.7112

Epoch 33:
Training loss: 1.0290, Training accuracy: 0.7066
Validation loss: 0.7925, Validation accuracy: 0.7268

Epoch 34:
Training loss: 1.0238, Training accuracy: 0.7110
Validation loss: 0.8062, Validation accuracy: 0.7192

Epoch 35:
Training loss: 1.0285, Training accuracy: 0.7102
Validation loss: 0.8106, Validation accuracy: 0.7250

Current learning rate has decayed to 0.006302
Epoch 36:
```

```
Training loss: 1.0231, Training accuracy: 0.7111
Validation loss: 0.7762, Validation accuracy: 0.7266

Epoch 37:
Training loss: 1.0206, Training accuracy: 0.7112
Validation loss: 0.7960, Validation accuracy: 0.7240

Epoch 38:
Training loss: 1.0224, Training accuracy: 0.7123
Validation loss: 0.7829, Validation accuracy: 0.7278

Epoch 39:
Training loss: 1.0238, Training accuracy: 0.7089
Validation loss: 0.7657, Validation accuracy: 0.7306

Current learning rate has decayed to 0.005987
Epoch 40:
Training loss: 1.0100, Training accuracy: 0.7164
Validation loss: 0.7733, Validation accuracy: 0.7266

Epoch 41:
Training loss: 1.0072, Training accuracy: 0.7168
Validation loss: 0.7629, Validation accuracy: 0.7352

Epoch 42:
Training loss: 1.0055, Training accuracy: 0.7194
Validation loss: 0.7642, Validation accuracy: 0.7328

Epoch 43:
Training loss: 1.0020, Training accuracy: 0.7182
Validation loss: 0.7700, Validation accuracy: 0.7284

Current learning rate has decayed to 0.005688
Epoch 44:
Training loss: 1.0022, Training accuracy: 0.7193
Validation loss: 0.7838, Validation accuracy: 0.7278

Epoch 45:
Training loss: 1.0006, Training accuracy: 0.7205
Validation loss: 0.7628, Validation accuracy: 0.7348

Epoch 46:
Training loss: 0.9984, Training accuracy: 0.7216
Validation loss: 0.7515, Validation accuracy: 0.7330

Epoch 47:
Training loss: 0.9948, Training accuracy: 0.7221
Validation loss: 0.7685, Validation accuracy: 0.7332
```

```
Current learning rate has decayed to 0.005404
Epoch 48:
Training loss: 0.9892, Training accuracy: 0.7252
Validation loss: 0.7630, Validation accuracy: 0.7338

Epoch 49:
Training loss: 0.9822, Training accuracy: 0.7244
Validation loss: 0.7835, Validation accuracy: 0.7330

Epoch 50:
Training loss: 0.9911, Training accuracy: 0.7224
Validation loss: 0.7681, Validation accuracy: 0.7308

Epoch 51:
Training loss: 0.9908, Training accuracy: 0.7228
Validation loss: 0.7556, Validation accuracy: 0.7384

Current learning rate has decayed to 0.005133
Epoch 52:
Training loss: 0.9860, Training accuracy: 0.7232
Validation loss: 0.7491, Validation accuracy: 0.7406

Epoch 53:
Training loss: 0.9856, Training accuracy: 0.7264
Validation loss: 0.7824, Validation accuracy: 0.7292

Epoch 54:
Training loss: 0.9798, Training accuracy: 0.7252
Validation loss: 0.7358, Validation accuracy: 0.7430

Epoch 55:
Training loss: 0.9792, Training accuracy: 0.7276
Validation loss: 0.7205, Validation accuracy: 0.7514

Current learning rate has decayed to 0.004877
Epoch 56:
Training loss: 0.9701, Training accuracy: 0.7299
Validation loss: 0.7431, Validation accuracy: 0.7396

Epoch 57:
Training loss: 0.9727, Training accuracy: 0.7292
Validation loss: 0.7261, Validation accuracy: 0.7470

Epoch 58:
Training loss: 0.9739, Training accuracy: 0.7277
Validation loss: 0.7697, Validation accuracy: 0.7300
```

```
Epoch 59:
Training loss: 0.9648, Training accuracy: 0.7328
Validation loss: 0.7259, Validation accuracy: 0.7484

Current learning rate has decayed to 0.004633
Epoch 60:
Training loss: 0.9671, Training accuracy: 0.7308
Validation loss: 0.7408, Validation accuracy: 0.7424

Epoch 61:
Training loss: 0.9649, Training accuracy: 0.7318
Validation loss: 0.7314, Validation accuracy: 0.7460

Epoch 62:
Training loss: 0.9590, Training accuracy: 0.7319
Validation loss: 0.7318, Validation accuracy: 0.7452

Epoch 63:
Training loss: 0.9603, Training accuracy: 0.7374
Validation loss: 0.7345, Validation accuracy: 0.7440

Current learning rate has decayed to 0.004401
Epoch 64:
Training loss: 0.9530, Training accuracy: 0.7343
Validation loss: 0.7230, Validation accuracy: 0.7474

Epoch 65:
Training loss: 0.9533, Training accuracy: 0.7348
Validation loss: 0.7517, Validation accuracy: 0.7432

Epoch 66:
Training loss: 0.9600, Training accuracy: 0.7340
Validation loss: 0.7132, Validation accuracy: 0.7538

Epoch 67:
Training loss: 0.9560, Training accuracy: 0.7363
Validation loss: 0.7211, Validation accuracy: 0.7506

Current learning rate has decayed to 0.004181
Epoch 68:
Training loss: 0.9528, Training accuracy: 0.7342
Validation loss: 0.7118, Validation accuracy: 0.7554

Epoch 69:
Training loss: 0.9525, Training accuracy: 0.7361
Validation loss: 0.7406, Validation accuracy: 0.7460

Epoch 70:
```

```
Training loss: 0.9493, Training accuracy: 0.7349
Validation loss: 0.7211, Validation accuracy: 0.7474

Epoch 71:
Training loss: 0.9498, Training accuracy: 0.7357
Validation loss: 0.7193, Validation accuracy: 0.7442

Current learning rate has decayed to 0.003972
Epoch 72:
Training loss: 0.9447, Training accuracy: 0.7386
Validation loss: 0.7206, Validation accuracy: 0.7408

Epoch 73:
Training loss: 0.9417, Training accuracy: 0.7379
Validation loss: 0.7402, Validation accuracy: 0.7436

Epoch 74:
Training loss: 0.9449, Training accuracy: 0.7374
Validation loss: 0.7181, Validation accuracy: 0.7484

Epoch 75:
Training loss: 0.9415, Training accuracy: 0.7397
Validation loss: 0.7116, Validation accuracy: 0.7556

Current learning rate has decayed to 0.003774
Epoch 76:
Training loss: 0.9375, Training accuracy: 0.7397
Validation loss: 0.7272, Validation accuracy: 0.7476

Epoch 77:
Training loss: 0.9413, Training accuracy: 0.7411
Validation loss: 0.7454, Validation accuracy: 0.7368

Epoch 78:
Training loss: 0.9380, Training accuracy: 0.7414
Validation loss: 0.7188, Validation accuracy: 0.7522

Epoch 79:
Training loss: 0.9357, Training accuracy: 0.7432
Validation loss: 0.7252, Validation accuracy: 0.7474

================================================
==> Optimization finished! Best validation accuracy: 0.7556
```

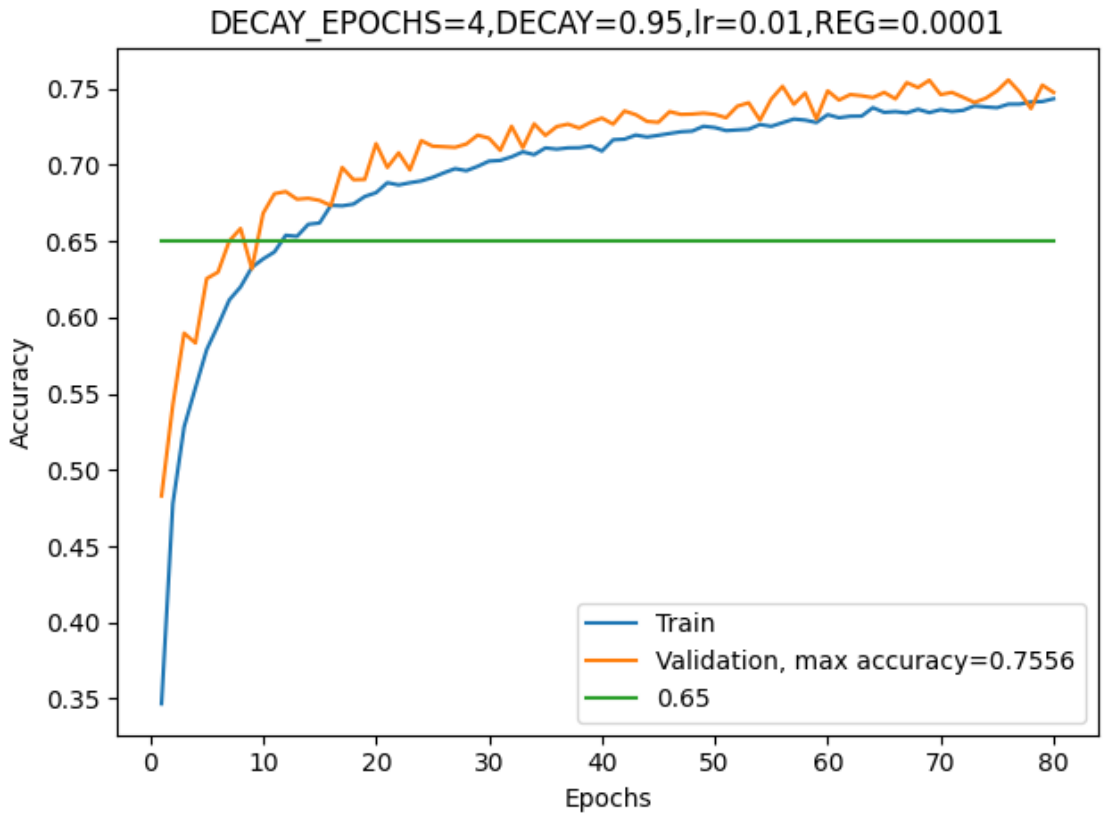
2 Bonus: with learning rate decay

The following code can help you adjust the learning rate during training. You need to figure out how to incorporate this code into your training loop.

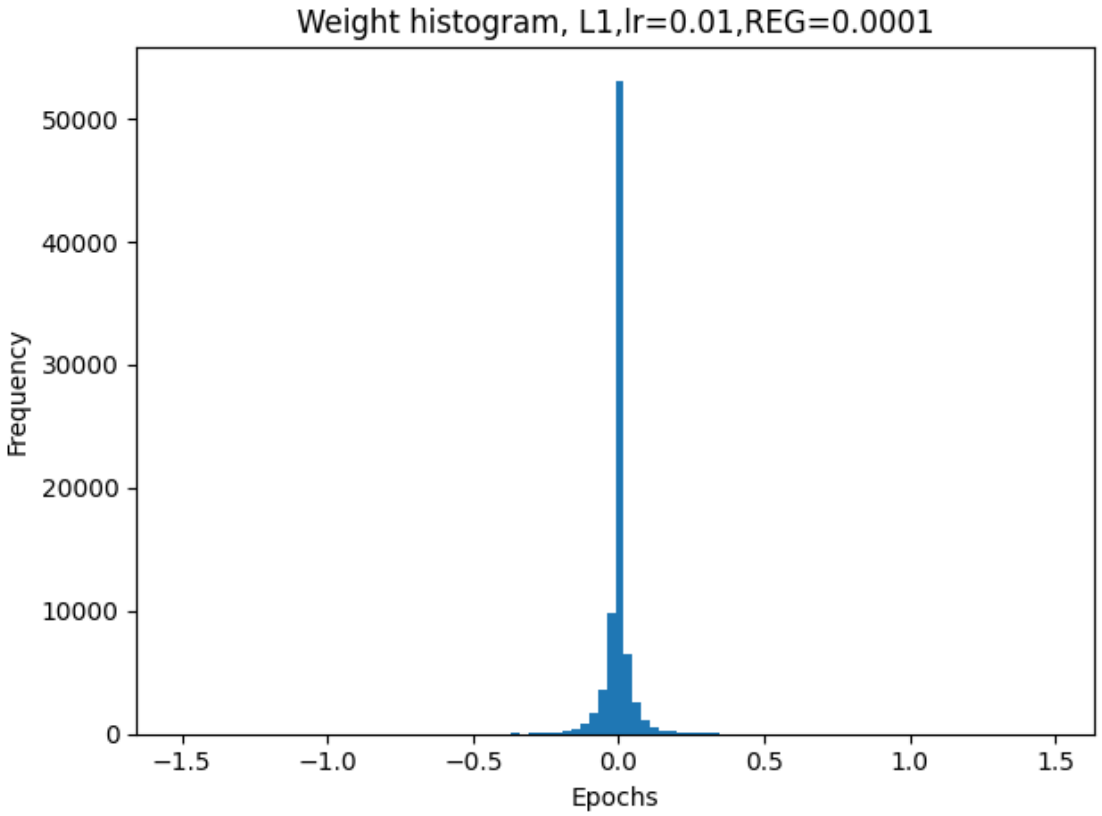
```
    if i % DECAY_EPOCHS == 0 and i != 0:
        current_learning_rate = current_learning_rate * DECAY
        for param_group in optimizer.param_groups:
            param_group['lr'] = current_learning_rate
        print("Current learning rate has decayed to %f" %current_learning_rate)
```

[11]:
```
##### plotting accuracy vs epoch


import matplotlib.pyplot as plt
import numpy as np

def my_to_np(lst):
    return np.array([lst[i].detach().cpu().numpy().flatten() for i in␣
 ↪range(len(lst))])

save = 'n'

fig, ax = plt.subplots(1, 1)
xx = np.linspace(1, EPOCHS, EPOCHS)

ax.plot(xx, my_to_np(accuracy_arr_train), label='Train')
ax.plot(xx, my_to_np(accuracy_arr_val), label='Validation, max accuracy={:.4f}'.
 ↪format(best_val_acc))
ax.plot(xx, np.linspace(0.65, 0.65, EPOCHS), label='0.65')
# title_ = 'DECAY_EPOCHS={:d}, DECAY={:g}, initial lr={:g}'.
 ↪format(DECAY_EPOCHS, DECAY, INITIAL_LR)
title_ = 'DECAY_EPOCHS={:d},DECAY={:g},lr={:g},REG={:g}'.format(DECAY_EPOCHS,␣
 ↪DECAY, INITIAL_LR, REG)
if DECAY == 1:
    title_ = 'No Learning Rate Decay'.format(DECAY_EPOCHS)
ax.set_xlabel('Epochs')
ax.set_ylabel('Accuracy')
ax.legend()
ax.set_title(title_)
fig.tight_layout()
if save == 'y':
    # plt.savefig('q2a_DECAY_EPOCHS_{:d}_DECAY_{:g}_aug.pdf'.
 ↪format(DECAY_EPOCHS, DECAY),
    #             dpi=500, bbox_inches='tight')
    # plt.savefig('q2b1_DECAY_EPOCHS_{:d}_DECAY_{:g}_aug_BN.pdf'.
 ↪format(DECAY_EPOCHS, DECAY),
    #             dpi=500, bbox_inches='tight')
```

```
  # plt.savefig('q2b2_aug_BN_lr{:g}.pdf'.format(INITIAL_LR),
  #             dpi=500, bbox_inches='tight')
  # plt.savefig('q2b2_aug_wo_BN_lr{:g}.pdf'.format(INITIAL_LR),
  #             dpi=500, bbox_inches='tight')
  # plt.savefig('q2b3_aug_BN_swish_lr{:g}_{:d}epochs.pdf'.format(INITIAL_LR,␣
↪EPOCHS),
  #             dpi=500, bbox_inches='tight')
  # plt.savefig('q2b3_aug_BN_relu_lr{:g}_{:d}epochs.pdf'.format(INITIAL_LR,␣
↪EPOCHS),
  #              dpi=500, bbox_inches='tight')
  # plt.savefig('q2c_aug_BN_swish_lr{:g}_{:d}epochs.pdf'.format(INITIAL_LR,␣
↪EPOCHS),
  #              dpi=500, bbox_inches='tight')
  # plt.savefig('q2c2_aug_BN_swish_lr{:g}_L2{:g}.pdf'.format(INITIAL_LR,␣
↪REG),
  #              dpi=500, bbox_inches='tight')
  plt.savefig('q2c3_aug_BN_swish_lr{:g}_{:s}_{:g}.pdf'.format(INITIAL_LR,␣
↪L1_or_L2, REG),
              dpi=500, bbox_inches='tight')
```



[12]:
```
##### plotting weights distribution

save = 'n'
fig, ax = plt.subplots(1, 1)

weights_list = []

for name, module in model.named_modules():
    if isinstance(module, nn.Conv2d) or isinstance(module, nn.Linear):
        weights_list.extend(list((module.weight.cpu().detach().numpy()).
 ↪flatten()))

ax.hist(weights_list, bins=100)
ax.set_title(str("Weight histogram, " + L1_or_L2 + ",lr={:g},REG={:g}".
 ↪format(INITIAL_LR, REG)))

ax.set_xlabel('Epochs')
ax.set_ylabel('Frequency')
fig.tight_layout()
if save == 'y':
    plt.savefig('q2c3weights_aug_BN_swish_lr{:g}_{:s}_{:g}.pdf'.
 ↪format(INITIAL_LR, L1_or_L2, REG),
                dpi=500, bbox_inches='tight')
```



resnet-cifar10

October 3, 2023

1 Training SimpleNN on CIFAR-10

In this project, you will use the SimpleNN model to perform image classification on CIFAR-10. CIFAR-10 orginally contains 60K images from 10 categories. We split it into 45K/5K/10K images to serve as train/valiation/test set. We only release the ground-truth labels of training/validation dataset to you.

1.1 Step 0: Set up the SimpleNN model

As you have practiced to implement simple neural networks in Homework 1, we just prepare the implementation for you.

[ ]:
```
# import necessary dependencies
import argparse
import os, sys
import time
import datetime
from tqdm import tqdm_notebook as tqdm

import torch
import torch.nn as nn
import torch.nn.functional as F
```

[2]:
```
img1 = torch.zeros((128, 8, 99, 99))
img2 = torch.zeros(img1.size())
img3 = torch.cat((img1, img2), 1)
print(img3.size())
```

```
torch.Size([128, 16, 99, 99])
```

[3]:
```
def my_swish(x):
    return torch.sigmoid(x) * x
## implementing swish


class Residual(nn.Module):
    def __init__(self, in_channels, out_channels, stride):
        """
```

```
        :param in_channels: number of channels for the input
        :param out_channels: number of channels for the output
        :param stride: stride of first conv2d block
        """
        # conv1: downsamples feature map when stride != 1 (padding = 1 to␣
↪ensure correct shape),
        # batch norm then ReLU
        super(Residual, self).__init__()
        self.conv1 = nn.Sequential(nn.Conv2d(in_channels, out_channels, 3,␣
↪stride=stride, padding=1),
                                   nn.BatchNorm2d(num_features=out_channels,␣
↪eps=1e-05, momentum=0.1),
                                   nn.ReLU())
        # conv2: doesn't downsample feature map since already performed in self.
↪conv1, batch norm,
        # padding=1 to ensure correct output shape (conv2 outputs same shape as␣
↪conv1's output)
        self.conv2 = nn.Sequential(nn.Conv2d(out_channels, out_channels, 3,␣
↪stride=1, padding=1),
                                   nn.BatchNorm2d(num_features=out_channels,␣
↪eps=1e-05, momentum=0.1))
        self.relu2 = nn.ReLU()  # applied after adding x in the forward function

        # in case we need these
        self.in_channels = in_channels
        self.out_channels = out_channels

    def forward(self, x):
        out = self.conv1(x)
        out = self.conv2(out)

        # if number of channels for input/output mismatch, must downsample
        # if x's num_channels doesn't match out's desired n_channels:
        if x.size(dim=1) != self.out_channels:
            x = x[:, :, ::2, ::2]  # downsample feature map by a factor of 2
            # following option (A), concatenate a zeros to increase n_channels␣
↪(bottom-left page776 of IEEE version).
            zzzzz = torch.zeros(x.size(), device=x.device)  # get zeros of same␣
↪size as x
            x_new = torch.cat((x, zzzzz), 1)  # concatenate channels
        else:
            x_new = x

        assert out.size() == x_new.size(), "out.size=={:s} != x.size=={:s}".
↪format(str(list(out.size())), str(list(x.size())))
        return self.relu2(out + x_new)
```

```
class ResNet(nn.Module):
    def __init__(self):
        super(ResNet, self).__init__()
        # implementing ResNet-20, so n=3
        # first conv to get 16-chan output, then (6x conv(in_channel=32), 6x␣
↪conv(in_channel=16),
        # 6x conv(in_channel=8)), then average pooling, then FC, finally softmax
        # maintain feature map size, increase n_channels to 16 after 1st␣
↪convolution
        self.conv_initial = nn.Sequential(nn.Conv2d(3, 16, 3, stride=1,␣
↪padding=1),
                                          nn.BatchNorm2d(num_features=16,␣
↪eps=1e-05, momentum=0.1),
                                          nn.ReLU())  # output 32*32 feature map
        # no max pool in this ResNet

        # first six conv layers (ReLU, batchNorm are applied within Residual)
        self.conv32_0 = Residual(in_channels=16, out_channels=16, stride=1) #␣
↪output 32*32 feature map
        self.conv32_2 = Residual(in_channels=16, out_channels=16, stride=1) #␣
↪output 32*32 feature map
        self.conv32_4 = Residual(in_channels=16, out_channels=16, stride=1) #␣
↪output 32*32 feature map

        # feature map is 16*16 after conv16_0
        # next six conv layers, (ReLU, batchNorm are applied within Residual)
        self.conv16_0 = Residual(in_channels=16, out_channels=32, stride=2) #␣
↪output 16*16 feature map
        self.conv16_2 = Residual(in_channels=32, out_channels=32, stride=1) #␣
↪output 16*16 feature map
        self.conv16_4 = Residual(in_channels=32, out_channels=32, stride=1) #␣
↪output 16*16 feature map

        # feature map decreases to 8*8 after conv8_0
        # last six conv layers, (ReLU, batchNorm are applied within Residual)
        self.conv8_0 = Residual(in_channels=32, out_channels=64, stride=2) #␣
↪output 8*8 feature map
        self.conv8_2 = Residual(in_channels=64, out_channels=64, stride=1) #␣
↪output 8*8 feature map
        self.conv8_4 = Residual(in_channels=64, out_channels=64, stride=1) #␣
↪output 8*8 feature map
        # output after self.conv8_4 should be 64*8*8

        # global average pooling (64*(8*8) -> 64*(1))
```

```
        self.glbal_avg_pool = nn.AvgPool2d(kernel_size=8)

        # fully-conneted layer, in_channels=64, out_channels=10
        self.fc = nn.Sequential(nn.Linear(64, 10))


    def forward(self, x):
        out = self.conv_initial(x)
        out = self.conv32_0(out)
        out = self.conv32_2(out)
        out = self.conv32_4(out)

        out = self.conv16_0(out)
        out = self.conv16_2(out)
        out = self.conv16_4(out)

        out = self.conv8_0(out)
        out = self.conv8_2(out)
        out = self.conv8_4(out)

        out = self.glbal_avg_pool(out)

        out = out.view(out.size(0), -1)  # flatten before FC
        out = self.fc(out)
        return out
```

1.1.1 Question (a)

Here is a sanity check to verify the implementation of SimpleNN. You need to: 1. Write down your code. 2. **In the PDF report**, give a brief description on how the code helps you know that SimpleNN is implemented correctly.

[4]:
```
################################################
# your code here
# sanity check for the correctness of ResNet
test_in = torch.rand((1, 3, 32, 32))
test_model = ResNet()
test_out = test_model.forward(test_in)
print(test_out.shape)
################################################
```

```
torch.Size([1, 10])
```

1.2 Step 1: Set up preprocessing functions

Preprocessing is very important as discussed in the lecture. You will need to write preprocessing functions with the help of *torchvision.transforms* in this step. You can find helpful tutorial/API at here.

1.2.1 Question (b)

For the question, you need to: 1. Complete the preprocessing code below. 2. **In the PDF report**, briefly describe what preprocessing operations you used and what are the purposes of them.

Hint: 1. Only two operations are necessary to complete the basic preprocessing here. 2. The raw input read from the dataset will be PIL images. 3. Data augmentation operations are not mendatory, but feel free to incorporate them if you want. 4. Reference value for mean/std of CIFAR-10 images (assuming the pixel values are within [0,1]): mean (RGB-format): (0.4914, 0.4822, 0.4465), std (RGB-format): (0.2023, 0.1994, 0.2010)

[5]:
```
# useful libraries
import torchvision
import torchvision.transforms as transforms

################################################
# your code here
# specify preprocessing function ToTensor (PIL to tensor) and Normalise (mean␣
 ↪becomes 0, stdev becomes 1)
# please see pdf for more details

transform_train =  transforms.Compose([
    transforms.ToTensor(),  ## data augmentation below for training only
    transforms.RandomCrop(size=(32, 32), padding=4, pad_if_needed=False,␣
 ↪padding_mode='edge'),
    transforms.RandomHorizontalFlip(p=0.5),
    transforms.Normalize((0.4914, 0.4822, 0.4465), (0.2023, 0.1994, 0.2010))
    ])

# validation set: no data augmentation
transform_val = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.4914, 0.4822, 0.4465), (0.2023, 0.1994, 0.2010))
    ])
################################################
```

1.3 Step 2: Set up dataset and dataloader

1.3.1 Question (c)

Set up the train/val datasets and dataloders that are to be used during the training. Check out the official API for more information about **torch.utils.data.DataLoader**.

Here, you need to: 1. Complete the code below.

[6]:
```
# do NOT change these
from tools.dataset import CIFAR10
from torch.utils.data import DataLoader
```

```
# a few arguments, do NOT change these
DATA_ROOT = "./data"
TRAIN_BATCH_SIZE = 128
VAL_BATCH_SIZE = 100

################################################
# your code here
# construct dataset
train_set = CIFAR10(
    root=DATA_ROOT,
    mode='train',
    download=True,
    transform=transform_train    # your code
)
val_set = CIFAR10(
    root=DATA_ROOT,
    mode='val',
    download=True,
    transform=transform_val    # your code
)

# construct dataloader
train_loader = DataLoader(
    train_set,
    batch_size=TRAIN_BATCH_SIZE,  # your code
    shuffle=True,     # your code
    num_workers=4
)
val_loader = DataLoader(
    val_set,
    batch_size=VAL_BATCH_SIZE,  # your code
    shuffle=True,     # your code
    num_workers=4
)
################################################
```

```
Using downloaded and verified file: ./data/cifar10_trainval_F22.zip
Extracting ./data/cifar10_trainval_F22.zip to ./data
Files already downloaded and verified
Using downloaded and verified file: ./data/cifar10_trainval_F22.zip
Extracting ./data/cifar10_trainval_F22.zip to ./data
Files already downloaded and verified
```

1.4 Step 3: Instantiate your SimpleNN model and deploy it to GPU devices.

1.4.1 Question (d)

You may want to deploy your model to GPU device for efficient training. Please assign your model to GPU if possible. If you are training on a machine without GPUs, please deploy your model to CPUs.

Here, you need to: 1. Complete the code below. 2. **In the PDF report**, briefly describe how you verify that your model is indeed deployed on GPU. (Hint: check `nvidia-smi`.)

[7]:
```
# specify the device for computation
#############################################
# your code here

device = 'cuda' if torch.cuda.is_available() else 'cpu'
print(device)
# Construct our model by instantiating the class defined above
model = ResNet()
# Copy to CUDA device. This is very important.
model = model.to(device)
#############################################
```

```
cuda
```

1.5 Step 4: Set up the loss function and optimizer

Loss function/objective function is used to provide "feedback" for the neural networks. Typically, we use multi-class cross-entropy as the loss function for classification models. As for the optimizer, we will use SGD with momentum.

1.5.1 Question (e)

Here, you need to: 1. Set up the cross-entropy loss as the criterion. (Hint: there are implemented functions in **torch.nn**) 2. Specify a SGD optimizer with momentum. (Hint: there are implemented functions in **torch.optim**)

[8]:
```
import torch.nn as nn
import torch.optim as optim



##### my addition
L1_or_L2 = 'L2'
##### end of my addition



# hyperparameters, do NOT change right now
# initial learning rate
```

```
INITIAL_LR = 0.1

# momentum for optimizer
MOMENTUM = 0.9

# L2 regularization strength, 1e-4 in ResNet paper
REG = 1e-4  # 1e-4 is orig reg strength (L2 and L1)

###############################################
# your code here
# create loss function
criterion = nn.CrossEntropyLoss()

# Add optimizer
assert MOMENTUM == 0.9, "momentum isn't 0.9"

# using L2 optimisation
if L1_or_L2 == 'L2':
    optimizer = torch.optim.SGD(model.parameters(), lr=INITIAL_LR,␣
 ↪momentum=MOMENTUM, weight_decay=REG)

# using L1 optimisation, weight_decay=0 (no L2)
if L1_or_L2 == 'L1':
    optimizer = torch.optim.SGD(model.parameters(), lr=INITIAL_LR,␣
 ↪momentum=MOMENTUM, weight_decay=0)

assert L1_or_L2 == 'L1' or L1_or_L2 == 'L2', 'L1_or_L2 should be either str L1␣
 ↪or L2'
###############################################
```

1.6 Step 5: Start the training process.

1.6.1 Question (f)/(g)

Congratulations! You have completed all of the previous steps and it is time to train our neural network.

Here you need to: 1. Complete the training codes. 2. Actually perform the training.

Hint: Training a neural network usually repeats the following 4 steps:

i) Get a batch of data from the dataloader and copy it to your device (GPU).

ii) Do a forward pass to get the outputs from the neural network and compute the loss. Be careful about your inputs to the loss function. Are the inputs required to be the logits or softmax probabilities?)

iii) Do a backward pass (back-propagation) to compute gradients of all weights with respect to the loss.

iiii) Update the model weights with the optimizer.

You will also need to compute the accuracy of training/validation samples to track your model's performance over each epoch (the accuracy should be increasing as you train for more and more epochs).

[9]:
```
# some hyperparameters
# total number of training epochs
EPOCHS = 200

# DECAY_EPOCHS = 4  # disregarded for ResNet
DECAY = 0.1  # large decay, happens at 32k iterations and 48k iterations

# the folder where the trained model is saved
CHECKPOINT_FOLDER = "./saved_model"

# start the training/validation process
# the process should take about 5 minutes on a GTX 1070-Ti
# if the code is written efficiently.
best_val_acc = 0
current_learning_rate = INITIAL_LR



##### my addition
accuracy_arr_train = []
accuracy_arr_val = []
num_iterations = 0
# keeps track of the number of iterations (incremented each mini-batch)
##### end of my addition




print("==> Training starts!")
print("="*50)
for i in range(0, EPOCHS):
    # handle the learning rate scheduler.
    if i == 100 or i == 150:  # at around 32k, 48k iterations
        current_learning_rate = current_learning_rate * DECAY
        for param_group in optimizer.param_groups:
            param_group['lr'] = current_learning_rate
        print("Current learning rate has decayed to %f" %current_learning_rate)
    else:
        print("Current learning rate is %f" %current_learning_rate)
    if num_iterations > 64000:
        print("According to ResNet paper, should terminate")
```

```
    ########################
    # your code here
    # switch to train mode
    model.train()

    ########################

    print("Epoch %d, iteration %d:" %(i, num_iterations))
    # this help you compute the training accuracy
    total_examples = 0
    correct_examples = 0

    train_loss = 0 # track training loss if you want

    # Train the model for 1 epoch.
    for batch_idx, (inputs, targets) in enumerate(train_loader):

        ####################################
        # your code here
        # copy inputs to device
        inputs, targets = inputs.to(device), targets.to(device)

        # compute the output and loss
        y_pred = model(inputs)
        loss = criterion(y_pred, targets)



        ## L1 normalisation
##### my addition
        if L1_or_L2 == 'L1':
            for name, param in model.named_parameters():
                if 'bias' not in name:
                    loss += REG * torch.sum(torch.abs(param))
        num_iterations += 1
##### end of my addition




        train_loss += loss

        # zero the gradient
        optimizer.zero_grad()

        # backpropagation
```

```
        loss.backward()

        # apply gradient and update the weights
        optimizer.step()

        # count the number of correctly predicted samples in the current batch
        total_examples += len(targets.view(targets.size(0), -1))
        correct_examples += sum(torch.argmax(y_pred, dim=1, keepdim=False) ==␣
 ↪targets)

        ###################################

    avg_loss = train_loss / len(train_loader)
    avg_acc = correct_examples / total_examples
    print("Training loss: %.4f, Training accuracy: %.4f" %(avg_loss, avg_acc))



##### my addition
    accuracy_arr_train.append(avg_acc)
##### end of my addition



    # Validate on the validation dataset
    ########################
    # your code here
    # switch to eval mode
    model.eval()

    ########################

    # this helps you compute the validation accuracy
    total_examples = 0
    correct_examples = 0

    val_loss = 0 # again, track the validation loss if you want

    # disable gradient during validation, which can save GPU memory
    with torch.no_grad():
        for batch_idx, (inputs, targets) in enumerate(val_loader):
            ###################################
            # your code here
            # copy inputs to device
            inputs, targets = inputs.to(device), targets.to(device)

            # compute the output and loss
```

```
            y_pred = model(inputs)
            loss = criterion(y_pred, targets)
            val_loss += loss

            # count the number of correctly predicted samples in the current␣
 ↪batch
            total_examples += len(targets.view(targets.size(0), -1))
            correct_examples += sum(torch.argmax(y_pred, dim=1, keepdim=False)␣
 ↪== targets)
            ######################################

    avg_loss = val_loss / len(val_loader)
    avg_acc = correct_examples / total_examples
    print("Validation loss: %.4f, Validation accuracy: %.4f" % (avg_loss,␣
 ↪avg_acc))



##### my addition
    accuracy_arr_val.append(avg_acc)
##### end of my addition



    # save the model checkpoint
    if avg_acc > best_val_acc:
        best_val_acc = avg_acc
        if not os.path.exists(CHECKPOINT_FOLDER):
           os.makedirs(CHECKPOINT_FOLDER)
        print("Saving ...")
        state = {'state_dict': model.state_dict(),
                'epoch': i,
                'lr': current_learning_rate}
        torch.save(state, os.path.join(CHECKPOINT_FOLDER, 'resnet-cifar10.pth'))

    print('')

print("="*50)
print(f"==> Optimization finished! Best validation accuracy: {best_val_acc:.
 ↪4f}")
```

```
==> Training starts!
==================================================
Current learning rate is 0.100000
Epoch 0, iteration 0:
Training loss: 1.6476, Training accuracy: 0.3938
Validation loss: 1.3719, Validation accuracy: 0.4918
```

```
Saving …

Current learning rate is 0.100000
Epoch 1, iteration 352:
Training loss: 1.1589, Training accuracy: 0.5844
Validation loss: 1.0514, Validation accuracy: 0.6222
Saving …

Current learning rate is 0.100000
Epoch 2, iteration 704:
Training loss: 0.9249, Training accuracy: 0.6746
Validation loss: 0.8934, Validation accuracy: 0.6948
Saving …

Current learning rate is 0.100000
Epoch 3, iteration 1056:
Training loss: 0.7827, Training accuracy: 0.7261
Validation loss: 0.7777, Validation accuracy: 0.7262
Saving …

Current learning rate is 0.100000
Epoch 4, iteration 1408:
Training loss: 0.6912, Training accuracy: 0.7585
Validation loss: 0.7061, Validation accuracy: 0.7518
Saving …

Current learning rate is 0.100000
Epoch 5, iteration 1760:
Training loss: 0.6258, Training accuracy: 0.7836
Validation loss: 0.6088, Validation accuracy: 0.7936
Saving …

Current learning rate is 0.100000
Epoch 6, iteration 2112:
Training loss: 0.5774, Training accuracy: 0.7980
Validation loss: 0.6128, Validation accuracy: 0.7936

Current learning rate is 0.100000
Epoch 7, iteration 2464:
Training loss: 0.5428, Training accuracy: 0.8109
Validation loss: 0.5466, Validation accuracy: 0.8134
Saving …

Current learning rate is 0.100000
Epoch 8, iteration 2816:
Training loss: 0.5113, Training accuracy: 0.8211
Validation loss: 0.6246, Validation accuracy: 0.7888
```

```
Current learning rate is 0.100000
Epoch 9, iteration 3168:
Training loss: 0.4843, Training accuracy: 0.8334
Validation loss: 0.4873, Validation accuracy: 0.8278
Saving …

Current learning rate is 0.100000
Epoch 10, iteration 3520:
Training loss: 0.4597, Training accuracy: 0.8394
Validation loss: 0.5629, Validation accuracy: 0.8092

Current learning rate is 0.100000
Epoch 11, iteration 3872:
Training loss: 0.4458, Training accuracy: 0.8474
Validation loss: 0.5632, Validation accuracy: 0.8134

Current learning rate is 0.100000
Epoch 12, iteration 4224:
Training loss: 0.4282, Training accuracy: 0.8507
Validation loss: 0.5131, Validation accuracy: 0.8234

Current learning rate is 0.100000
Epoch 13, iteration 4576:
Training loss: 0.4107, Training accuracy: 0.8583
Validation loss: 0.5392, Validation accuracy: 0.8210

Current learning rate is 0.100000
Epoch 14, iteration 4928:
Training loss: 0.3986, Training accuracy: 0.8629
Validation loss: 0.5402, Validation accuracy: 0.8196

Current learning rate is 0.100000
Epoch 15, iteration 5280:
Training loss: 0.3914, Training accuracy: 0.8659
Validation loss: 0.5059, Validation accuracy: 0.8318
Saving …

Current learning rate is 0.100000
Epoch 16, iteration 5632:
Training loss: 0.3780, Training accuracy: 0.8682
Validation loss: 0.4649, Validation accuracy: 0.8424
Saving …

Current learning rate is 0.100000
Epoch 17, iteration 5984:
Training loss: 0.3703, Training accuracy: 0.8724
Validation loss: 0.6468, Validation accuracy: 0.8032
```

```
Current learning rate is 0.100000
Epoch 18, iteration 6336:
Training loss: 0.3618, Training accuracy: 0.8757
Validation loss: 0.5237, Validation accuracy: 0.8238

Current learning rate is 0.100000
Epoch 19, iteration 6688:
Training loss: 0.3534, Training accuracy: 0.8772
Validation loss: 0.4907, Validation accuracy: 0.8362

Current learning rate is 0.100000
Epoch 20, iteration 7040:
Training loss: 0.3472, Training accuracy: 0.8803
Validation loss: 0.4831, Validation accuracy: 0.8488
Saving …

Current learning rate is 0.100000
Epoch 21, iteration 7392:
Training loss: 0.3412, Training accuracy: 0.8816
Validation loss: 0.4594, Validation accuracy: 0.8468

Current learning rate is 0.100000
Epoch 22, iteration 7744:
Training loss: 0.3308, Training accuracy: 0.8868
Validation loss: 0.4972, Validation accuracy: 0.8378

Current learning rate is 0.100000
Epoch 23, iteration 8096:
Training loss: 0.3245, Training accuracy: 0.8872
Validation loss: 0.4584, Validation accuracy: 0.8500
Saving …

Current learning rate is 0.100000
Epoch 24, iteration 8448:
Training loss: 0.3205, Training accuracy: 0.8876
Validation loss: 0.4552, Validation accuracy: 0.8458

Current learning rate is 0.100000
Epoch 25, iteration 8800:
Training loss: 0.3131, Training accuracy: 0.8914
Validation loss: 0.4629, Validation accuracy: 0.8510
Saving …

Current learning rate is 0.100000
Epoch 26, iteration 9152:
Training loss: 0.3106, Training accuracy: 0.8928
Validation loss: 0.4233, Validation accuracy: 0.8588
Saving …
```

```
Current learning rate is 0.100000
Epoch 27, iteration 9504:
Training loss: 0.2971, Training accuracy: 0.8968
Validation loss: 0.5892, Validation accuracy: 0.8198

Current learning rate is 0.100000
Epoch 28, iteration 9856:
Training loss: 0.3003, Training accuracy: 0.8945
Validation loss: 0.4929, Validation accuracy: 0.8370

Current learning rate is 0.100000
Epoch 29, iteration 10208:
Training loss: 0.2937, Training accuracy: 0.8985
Validation loss: 0.5049, Validation accuracy: 0.8382

Current learning rate is 0.100000
Epoch 30, iteration 10560:
Training loss: 0.2949, Training accuracy: 0.8964
Validation loss: 0.4326, Validation accuracy: 0.8612
Saving …

Current learning rate is 0.100000
Epoch 31, iteration 10912:
Training loss: 0.2948, Training accuracy: 0.8971
Validation loss: 0.4329, Validation accuracy: 0.8570

Current learning rate is 0.100000
Epoch 32, iteration 11264:
Training loss: 0.2886, Training accuracy: 0.8990
Validation loss: 0.6566, Validation accuracy: 0.8082

Current learning rate is 0.100000
Epoch 33, iteration 11616:
Training loss: 0.2834, Training accuracy: 0.9009
Validation loss: 0.4657, Validation accuracy: 0.8558

Current learning rate is 0.100000
Epoch 34, iteration 11968:
Training loss: 0.2819, Training accuracy: 0.9012
Validation loss: 0.4616, Validation accuracy: 0.8544

Current learning rate is 0.100000
Epoch 35, iteration 12320:
Training loss: 0.2778, Training accuracy: 0.9026
Validation loss: 0.4908, Validation accuracy: 0.8464

Current learning rate is 0.100000
```

```
Epoch 36, iteration 12672:
Training loss: 0.2672, Training accuracy: 0.9059
Validation loss: 0.5385, Validation accuracy: 0.8338

Current learning rate is 0.100000
Epoch 37, iteration 13024:
Training loss: 0.2775, Training accuracy: 0.9024
Validation loss: 0.4091, Validation accuracy: 0.8672
Saving …

Current learning rate is 0.100000
Epoch 38, iteration 13376:
Training loss: 0.2672, Training accuracy: 0.9078
Validation loss: 0.4336, Validation accuracy: 0.8624

Current learning rate is 0.100000
Epoch 39, iteration 13728:
Training loss: 0.2627, Training accuracy: 0.9067
Validation loss: 0.4346, Validation accuracy: 0.8574

Current learning rate is 0.100000
Epoch 40, iteration 14080:
Training loss: 0.2597, Training accuracy: 0.9089
Validation loss: 0.4567, Validation accuracy: 0.8488

Current learning rate is 0.100000
Epoch 41, iteration 14432:
Training loss: 0.2596, Training accuracy: 0.9079
Validation loss: 0.5418, Validation accuracy: 0.8350

Current learning rate is 0.100000
Epoch 42, iteration 14784:
Training loss: 0.2607, Training accuracy: 0.9103
Validation loss: 0.4263, Validation accuracy: 0.8648

Current learning rate is 0.100000
Epoch 43, iteration 15136:
Training loss: 0.2519, Training accuracy: 0.9109
Validation loss: 0.4542, Validation accuracy: 0.8566

Current learning rate is 0.100000
Epoch 44, iteration 15488:
Training loss: 0.2496, Training accuracy: 0.9129
Validation loss: 0.5789, Validation accuracy: 0.8372

Current learning rate is 0.100000
Epoch 45, iteration 15840:
Training loss: 0.2551, Training accuracy: 0.9118
```

```
Validation loss: 0.4412, Validation accuracy: 0.8602

Current learning rate is 0.100000
Epoch 46, iteration 16192:
Training loss: 0.2500, Training accuracy: 0.9134
Validation loss: 0.4825, Validation accuracy: 0.8566

Current learning rate is 0.100000
Epoch 47, iteration 16544:
Training loss: 0.2503, Training accuracy: 0.9123
Validation loss: 0.4366, Validation accuracy: 0.8614

Current learning rate is 0.100000
Epoch 48, iteration 16896:
Training loss: 0.2488, Training accuracy: 0.9136
Validation loss: 0.4049, Validation accuracy: 0.8766
Saving …

Current learning rate is 0.100000
Epoch 49, iteration 17248:
Training loss: 0.2387, Training accuracy: 0.9164
Validation loss: 0.3991, Validation accuracy: 0.8666

Current learning rate is 0.100000
Epoch 50, iteration 17600:
Training loss: 0.2408, Training accuracy: 0.9153
Validation loss: 0.4733, Validation accuracy: 0.8516

Current learning rate is 0.100000
Epoch 51, iteration 17952:
Training loss: 0.2386, Training accuracy: 0.9161
Validation loss: 0.4781, Validation accuracy: 0.8494

Current learning rate is 0.100000
Epoch 52, iteration 18304:
Training loss: 0.2369, Training accuracy: 0.9173
Validation loss: 0.5785, Validation accuracy: 0.8228

Current learning rate is 0.100000
Epoch 53, iteration 18656:
Training loss: 0.2353, Training accuracy: 0.9176
Validation loss: 0.4095, Validation accuracy: 0.8636

Current learning rate is 0.100000
Epoch 54, iteration 19008:
Training loss: 0.2348, Training accuracy: 0.9180
Validation loss: 0.4701, Validation accuracy: 0.8624
```

```
Current learning rate is 0.100000
Epoch 55, iteration 19360:
Training loss: 0.2320, Training accuracy: 0.9192
Validation loss: 0.3708, Validation accuracy: 0.8794
Saving …

Current learning rate is 0.100000
Epoch 56, iteration 19712:
Training loss: 0.2373, Training accuracy: 0.9169
Validation loss: 0.4145, Validation accuracy: 0.8702

Current learning rate is 0.100000
Epoch 57, iteration 20064:
Training loss: 0.2289, Training accuracy: 0.9200
Validation loss: 0.4503, Validation accuracy: 0.8612

Current learning rate is 0.100000
Epoch 58, iteration 20416:
Training loss: 0.2309, Training accuracy: 0.9186
Validation loss: 0.4221, Validation accuracy: 0.8668

Current learning rate is 0.100000
Epoch 59, iteration 20768:
Training loss: 0.2253, Training accuracy: 0.9199
Validation loss: 0.3917, Validation accuracy: 0.8824
Saving …

Current learning rate is 0.100000
Epoch 60, iteration 21120:
Training loss: 0.2286, Training accuracy: 0.9192
Validation loss: 0.4906, Validation accuracy: 0.8536

Current learning rate is 0.100000
Epoch 61, iteration 21472:
Training loss: 0.2233, Training accuracy: 0.9223
Validation loss: 0.4167, Validation accuracy: 0.8696

Current learning rate is 0.100000
Epoch 62, iteration 21824:
Training loss: 0.2292, Training accuracy: 0.9190
Validation loss: 0.3802, Validation accuracy: 0.8742

Current learning rate is 0.100000
Epoch 63, iteration 22176:
Training loss: 0.2252, Training accuracy: 0.9218
Validation loss: 0.4094, Validation accuracy: 0.8662

Current learning rate is 0.100000
```

```
Epoch 64, iteration 22528:
Training loss: 0.2232, Training accuracy: 0.9221
Validation loss: 0.3788, Validation accuracy: 0.8808

Current learning rate is 0.100000
Epoch 65, iteration 22880:
Training loss: 0.2257, Training accuracy: 0.9193
Validation loss: 0.4606, Validation accuracy: 0.8618

Current learning rate is 0.100000
Epoch 66, iteration 23232:
Training loss: 0.2196, Training accuracy: 0.9222
Validation loss: 0.4055, Validation accuracy: 0.8708

Current learning rate is 0.100000
Epoch 67, iteration 23584:
Training loss: 0.2228, Training accuracy: 0.9223
Validation loss: 0.4834, Validation accuracy: 0.8498

Current learning rate is 0.100000
Epoch 68, iteration 23936:
Training loss: 0.2179, Training accuracy: 0.9221
Validation loss: 0.4973, Validation accuracy: 0.8464

Current learning rate is 0.100000
Epoch 69, iteration 24288:
Training loss: 0.2129, Training accuracy: 0.9265
Validation loss: 0.4101, Validation accuracy: 0.8666

Current learning rate is 0.100000
Epoch 70, iteration 24640:
Training loss: 0.2230, Training accuracy: 0.9216
Validation loss: 0.4022, Validation accuracy: 0.8674

Current learning rate is 0.100000
Epoch 71, iteration 24992:
Training loss: 0.2128, Training accuracy: 0.9252
Validation loss: 0.4079, Validation accuracy: 0.8678

Current learning rate is 0.100000
Epoch 72, iteration 25344:
Training loss: 0.2199, Training accuracy: 0.9238
Validation loss: 0.3891, Validation accuracy: 0.8724

Current learning rate is 0.100000
Epoch 73, iteration 25696:
Training loss: 0.2165, Training accuracy: 0.9250
Validation loss: 0.4776, Validation accuracy: 0.8600
```

```
Current learning rate is 0.100000
Epoch 74, iteration 26048:
Training loss: 0.2087, Training accuracy: 0.9266
Validation loss: 0.4351, Validation accuracy: 0.8650

Current learning rate is 0.100000
Epoch 75, iteration 26400:
Training loss: 0.2156, Training accuracy: 0.9234
Validation loss: 0.3953, Validation accuracy: 0.8760

Current learning rate is 0.100000
Epoch 76, iteration 26752:
Training loss: 0.2106, Training accuracy: 0.9254
Validation loss: 0.4357, Validation accuracy: 0.8636

Current learning rate is 0.100000
Epoch 77, iteration 27104:
Training loss: 0.2101, Training accuracy: 0.9257
Validation loss: 0.3806, Validation accuracy: 0.8824

Current learning rate is 0.100000
Epoch 78, iteration 27456:
Training loss: 0.2147, Training accuracy: 0.9242
Validation loss: 0.3566, Validation accuracy: 0.8860
Saving …

Current learning rate is 0.100000
Epoch 79, iteration 27808:
Training loss: 0.2064, Training accuracy: 0.9270
Validation loss: 0.4410, Validation accuracy: 0.8692

Current learning rate is 0.100000
Epoch 80, iteration 28160:
Training loss: 0.2137, Training accuracy: 0.9251
Validation loss: 0.4551, Validation accuracy: 0.8612

Current learning rate is 0.100000
Epoch 81, iteration 28512:
Training loss: 0.2063, Training accuracy: 0.9280
Validation loss: 0.3742, Validation accuracy: 0.8852

Current learning rate is 0.100000
Epoch 82, iteration 28864:
Training loss: 0.2073, Training accuracy: 0.9261
Validation loss: 0.4863, Validation accuracy: 0.8530

Current learning rate is 0.100000
```

```
Epoch 83, iteration 29216:
Training loss: 0.2105, Training accuracy: 0.9262
Validation loss: 0.4233, Validation accuracy: 0.8662

Current learning rate is 0.100000
Epoch 84, iteration 29568:
Training loss: 0.2086, Training accuracy: 0.9265
Validation loss: 0.5409, Validation accuracy: 0.8494

Current learning rate is 0.100000
Epoch 85, iteration 29920:
Training loss: 0.2063, Training accuracy: 0.9283
Validation loss: 0.4195, Validation accuracy: 0.8720

Current learning rate is 0.100000
Epoch 86, iteration 30272:
Training loss: 0.2083, Training accuracy: 0.9280
Validation loss: 0.4213, Validation accuracy: 0.8740

Current learning rate is 0.100000
Epoch 87, iteration 30624:
Training loss: 0.2039, Training accuracy: 0.9287
Validation loss: 0.4776, Validation accuracy: 0.8560

Current learning rate is 0.100000
Epoch 88, iteration 30976:
Training loss: 0.2027, Training accuracy: 0.9280
Validation loss: 0.4101, Validation accuracy: 0.8642

Current learning rate is 0.100000
Epoch 89, iteration 31328:
Training loss: 0.2036, Training accuracy: 0.9279
Validation loss: 0.3694, Validation accuracy: 0.8840

Current learning rate is 0.100000
Epoch 90, iteration 31680:
Training loss: 0.2117, Training accuracy: 0.9248
Validation loss: 0.4063, Validation accuracy: 0.8722

Current learning rate is 0.100000
Epoch 91, iteration 32032:
Training loss: 0.2066, Training accuracy: 0.9287
Validation loss: 0.4331, Validation accuracy: 0.8708

Current learning rate is 0.100000
Epoch 92, iteration 32384:
Training loss: 0.2039, Training accuracy: 0.9279
Validation loss: 0.3904, Validation accuracy: 0.8714
```

```
Current learning rate is 0.100000
Epoch 93, iteration 32736:
Training loss: 0.2016, Training accuracy: 0.9287
Validation loss: 0.4854, Validation accuracy: 0.8492

Current learning rate is 0.100000
Epoch 94, iteration 33088:
Training loss: 0.2007, Training accuracy: 0.9298
Validation loss: 0.3460, Validation accuracy: 0.8874
Saving …

Current learning rate is 0.100000
Epoch 95, iteration 33440:
Training loss: 0.1992, Training accuracy: 0.9312
Validation loss: 0.4527, Validation accuracy: 0.8542

Current learning rate is 0.100000
Epoch 96, iteration 33792:
Training loss: 0.2019, Training accuracy: 0.9288
Validation loss: 0.4418, Validation accuracy: 0.8680

Current learning rate is 0.100000
Epoch 97, iteration 34144:
Training loss: 0.1990, Training accuracy: 0.9310
Validation loss: 0.4205, Validation accuracy: 0.8710

Current learning rate is 0.100000
Epoch 98, iteration 34496:
Training loss: 0.2033, Training accuracy: 0.9297
Validation loss: 0.5346, Validation accuracy: 0.8458

Current learning rate is 0.100000
Epoch 99, iteration 34848:
Training loss: 0.1949, Training accuracy: 0.9316
Validation loss: 0.3870, Validation accuracy: 0.8798

Current learning rate has decayed to 0.010000
Epoch 100, iteration 35200:
Training loss: 0.1179, Training accuracy: 0.9612
Validation loss: 0.2675, Validation accuracy: 0.9126
Saving …

Current learning rate is 0.010000
Epoch 101, iteration 35552:
Training loss: 0.0881, Training accuracy: 0.9712
Validation loss: 0.2584, Validation accuracy: 0.9194
Saving …
```

```
Current learning rate is 0.010000
Epoch 102, iteration 35904:
Training loss: 0.0832, Training accuracy: 0.9722
Validation loss: 0.2571, Validation accuracy: 0.9196
Saving …

Current learning rate is 0.010000
Epoch 103, iteration 36256:
Training loss: 0.0737, Training accuracy: 0.9754
Validation loss: 0.2595, Validation accuracy: 0.9194

Current learning rate is 0.010000
Epoch 104, iteration 36608:
Training loss: 0.0675, Training accuracy: 0.9772
Validation loss: 0.2608, Validation accuracy: 0.9198
Saving …

Current learning rate is 0.010000
Epoch 105, iteration 36960:
Training loss: 0.0649, Training accuracy: 0.9783
Validation loss: 0.2576, Validation accuracy: 0.9214
Saving …

Current learning rate is 0.010000
Epoch 106, iteration 37312:
Training loss: 0.0615, Training accuracy: 0.9800
Validation loss: 0.2634, Validation accuracy: 0.9210

Current learning rate is 0.010000
Epoch 107, iteration 37664:
Training loss: 0.0587, Training accuracy: 0.9808
Validation loss: 0.2650, Validation accuracy: 0.9198

Current learning rate is 0.010000
Epoch 108, iteration 38016:
Training loss: 0.0541, Training accuracy: 0.9828
Validation loss: 0.2631, Validation accuracy: 0.9242
Saving …

Current learning rate is 0.010000
Epoch 109, iteration 38368:
Training loss: 0.0521, Training accuracy: 0.9828
Validation loss: 0.2634, Validation accuracy: 0.9246
Saving …

Current learning rate is 0.010000
Epoch 110, iteration 38720:
```

```
Training loss: 0.0508, Training accuracy: 0.9836
Validation loss: 0.2702, Validation accuracy: 0.9214

Current learning rate is 0.010000
Epoch 111, iteration 39072:
Training loss: 0.0489, Training accuracy: 0.9842
Validation loss: 0.2731, Validation accuracy: 0.9208

Current learning rate is 0.010000
Epoch 112, iteration 39424:
Training loss: 0.0480, Training accuracy: 0.9847
Validation loss: 0.2804, Validation accuracy: 0.9204

Current learning rate is 0.010000
Epoch 113, iteration 39776:
Training loss: 0.0461, Training accuracy: 0.9849
Validation loss: 0.2775, Validation accuracy: 0.9226

Current learning rate is 0.010000
Epoch 114, iteration 40128:
Training loss: 0.0448, Training accuracy: 0.9856
Validation loss: 0.2796, Validation accuracy: 0.9222

Current learning rate is 0.010000
Epoch 115, iteration 40480:
Training loss: 0.0435, Training accuracy: 0.9859
Validation loss: 0.2835, Validation accuracy: 0.9222

Current learning rate is 0.010000
Epoch 116, iteration 40832:
Training loss: 0.0403, Training accuracy: 0.9870
Validation loss: 0.2848, Validation accuracy: 0.9218

Current learning rate is 0.010000
Epoch 117, iteration 41184:
Training loss: 0.0417, Training accuracy: 0.9862
Validation loss: 0.2824, Validation accuracy: 0.9224

Current learning rate is 0.010000
Epoch 118, iteration 41536:
Training loss: 0.0397, Training accuracy: 0.9871
Validation loss: 0.2835, Validation accuracy: 0.9226

Current learning rate is 0.010000
Epoch 119, iteration 41888:
Training loss: 0.0366, Training accuracy: 0.9884
Validation loss: 0.2846, Validation accuracy: 0.9214
```

```
Current learning rate is 0.010000
Epoch 120, iteration 42240:
Training loss: 0.0369, Training accuracy: 0.9880
Validation loss: 0.2934, Validation accuracy: 0.9202

Current learning rate is 0.010000
Epoch 121, iteration 42592:
Training loss: 0.0364, Training accuracy: 0.9883
Validation loss: 0.2893, Validation accuracy: 0.9238

Current learning rate is 0.010000
Epoch 122, iteration 42944:
Training loss: 0.0357, Training accuracy: 0.9884
Validation loss: 0.3048, Validation accuracy: 0.9206

Current learning rate is 0.010000
Epoch 123, iteration 43296:
Training loss: 0.0345, Training accuracy: 0.9886
Validation loss: 0.3016, Validation accuracy: 0.9194

Current learning rate is 0.010000
Epoch 124, iteration 43648:
Training loss: 0.0331, Training accuracy: 0.9894
Validation loss: 0.3049, Validation accuracy: 0.9214

Current learning rate is 0.010000
Epoch 125, iteration 44000:
Training loss: 0.0302, Training accuracy: 0.9905
Validation loss: 0.3049, Validation accuracy: 0.9206

Current learning rate is 0.010000
Epoch 126, iteration 44352:
Training loss: 0.0325, Training accuracy: 0.9900
Validation loss: 0.3062, Validation accuracy: 0.9206

Current learning rate is 0.010000
Epoch 127, iteration 44704:
Training loss: 0.0311, Training accuracy: 0.9906
Validation loss: 0.3127, Validation accuracy: 0.9216

Current learning rate is 0.010000
Epoch 128, iteration 45056:
Training loss: 0.0308, Training accuracy: 0.9902
Validation loss: 0.3168, Validation accuracy: 0.9182

Current learning rate is 0.010000
Epoch 129, iteration 45408:
Training loss: 0.0301, Training accuracy: 0.9905
```

```
Validation loss: 0.2979, Validation accuracy: 0.9202

Current learning rate is 0.010000
Epoch 130, iteration 45760:
Training loss: 0.0303, Training accuracy: 0.9903
Validation loss: 0.3077, Validation accuracy: 0.9224

Current learning rate is 0.010000
Epoch 131, iteration 46112:
Training loss: 0.0282, Training accuracy: 0.9911
Validation loss: 0.3087, Validation accuracy: 0.9234

Current learning rate is 0.010000
Epoch 132, iteration 46464:
Training loss: 0.0270, Training accuracy: 0.9916
Validation loss: 0.3152, Validation accuracy: 0.9232

Current learning rate is 0.010000
Epoch 133, iteration 46816:
Training loss: 0.0270, Training accuracy: 0.9917
Validation loss: 0.3243, Validation accuracy: 0.9198

Current learning rate is 0.010000
Epoch 134, iteration 47168:
Training loss: 0.0261, Training accuracy: 0.9916
Validation loss: 0.3148, Validation accuracy: 0.9218

Current learning rate is 0.010000
Epoch 135, iteration 47520:
Training loss: 0.0262, Training accuracy: 0.9922
Validation loss: 0.3246, Validation accuracy: 0.9196

Current learning rate is 0.010000
Epoch 136, iteration 47872:
Training loss: 0.0261, Training accuracy: 0.9919
Validation loss: 0.3225, Validation accuracy: 0.9192

Current learning rate is 0.010000
Epoch 137, iteration 48224:
Training loss: 0.0258, Training accuracy: 0.9920
Validation loss: 0.3324, Validation accuracy: 0.9186

Current learning rate is 0.010000
Epoch 138, iteration 48576:
Training loss: 0.0251, Training accuracy: 0.9919
Validation loss: 0.3259, Validation accuracy: 0.9206

Current learning rate is 0.010000
```

```
Epoch 139, iteration 48928:
Training loss: 0.0228, Training accuracy: 0.9931
Validation loss: 0.3299, Validation accuracy: 0.9162

Current learning rate is 0.010000
Epoch 140, iteration 49280:
Training loss: 0.0250, Training accuracy: 0.9920
Validation loss: 0.3190, Validation accuracy: 0.9198

Current learning rate is 0.010000
Epoch 141, iteration 49632:
Training loss: 0.0244, Training accuracy: 0.9926
Validation loss: 0.3194, Validation accuracy: 0.9230

Current learning rate is 0.010000
Epoch 142, iteration 49984:
Training loss: 0.0229, Training accuracy: 0.9931
Validation loss: 0.3348, Validation accuracy: 0.9188

Current learning rate is 0.010000
Epoch 143, iteration 50336:
Training loss: 0.0230, Training accuracy: 0.9930
Validation loss: 0.3246, Validation accuracy: 0.9204

Current learning rate is 0.010000
Epoch 144, iteration 50688:
Training loss: 0.0243, Training accuracy: 0.9924
Validation loss: 0.3263, Validation accuracy: 0.9188

Current learning rate is 0.010000
Epoch 145, iteration 51040:
Training loss: 0.0227, Training accuracy: 0.9930
Validation loss: 0.3307, Validation accuracy: 0.9244

Current learning rate is 0.010000
Epoch 146, iteration 51392:
Training loss: 0.0230, Training accuracy: 0.9927
Validation loss: 0.3273, Validation accuracy: 0.9220

Current learning rate is 0.010000
Epoch 147, iteration 51744:
Training loss: 0.0197, Training accuracy: 0.9945
Validation loss: 0.3294, Validation accuracy: 0.9212

Current learning rate is 0.010000
Epoch 148, iteration 52096:
Training loss: 0.0220, Training accuracy: 0.9935
Validation loss: 0.3320, Validation accuracy: 0.9206
```

```
Current learning rate is 0.010000
Epoch 149, iteration 52448:
Training loss: 0.0211, Training accuracy: 0.9936
Validation loss: 0.3333, Validation accuracy: 0.9228

Current learning rate has decayed to 0.001000
Epoch 150, iteration 52800:
Training loss: 0.0177, Training accuracy: 0.9950
Validation loss: 0.3251, Validation accuracy: 0.9240

Current learning rate is 0.001000
Epoch 151, iteration 53152:
Training loss: 0.0180, Training accuracy: 0.9947
Validation loss: 0.3276, Validation accuracy: 0.9240

Current learning rate is 0.001000
Epoch 152, iteration 53504:
Training loss: 0.0165, Training accuracy: 0.9958
Validation loss: 0.3291, Validation accuracy: 0.9238

Current learning rate is 0.001000
Epoch 153, iteration 53856:
Training loss: 0.0159, Training accuracy: 0.9956
Validation loss: 0.3281, Validation accuracy: 0.9234

Current learning rate is 0.001000
Epoch 154, iteration 54208:
Training loss: 0.0161, Training accuracy: 0.9956
Validation loss: 0.3278, Validation accuracy: 0.9216

Current learning rate is 0.001000
Epoch 155, iteration 54560:
Training loss: 0.0155, Training accuracy: 0.9958
Validation loss: 0.3247, Validation accuracy: 0.9232

Current learning rate is 0.001000
Epoch 156, iteration 54912:
Training loss: 0.0151, Training accuracy: 0.9959
Validation loss: 0.3251, Validation accuracy: 0.9230

Current learning rate is 0.001000
Epoch 157, iteration 55264:
Training loss: 0.0151, Training accuracy: 0.9957
Validation loss: 0.3280, Validation accuracy: 0.9230

Current learning rate is 0.001000
Epoch 158, iteration 55616:
```

```
Training loss: 0.0148, Training accuracy: 0.9961
Validation loss: 0.3258, Validation accuracy: 0.9232

Current learning rate is 0.001000
Epoch 159, iteration 55968:
Training loss: 0.0146, Training accuracy: 0.9960
Validation loss: 0.3260, Validation accuracy: 0.9242

Current learning rate is 0.001000
Epoch 160, iteration 56320:
Training loss: 0.0151, Training accuracy: 0.9960
Validation loss: 0.3232, Validation accuracy: 0.9246

Current learning rate is 0.001000
Epoch 161, iteration 56672:
Training loss: 0.0138, Training accuracy: 0.9965
Validation loss: 0.3270, Validation accuracy: 0.9248
Saving …

Current learning rate is 0.001000
Epoch 162, iteration 57024:
Training loss: 0.0151, Training accuracy: 0.9960
Validation loss: 0.3233, Validation accuracy: 0.9242

Current learning rate is 0.001000
Epoch 163, iteration 57376:
Training loss: 0.0141, Training accuracy: 0.9964
Validation loss: 0.3228, Validation accuracy: 0.9248

Current learning rate is 0.001000
Epoch 164, iteration 57728:
Training loss: 0.0140, Training accuracy: 0.9963
Validation loss: 0.3243, Validation accuracy: 0.9232

Current learning rate is 0.001000
Epoch 165, iteration 58080:
Training loss: 0.0138, Training accuracy: 0.9964
Validation loss: 0.3230, Validation accuracy: 0.9242

Current learning rate is 0.001000
Epoch 166, iteration 58432:
Training loss: 0.0140, Training accuracy: 0.9969
Validation loss: 0.3270, Validation accuracy: 0.9234

Current learning rate is 0.001000
Epoch 167, iteration 58784:
Training loss: 0.0140, Training accuracy: 0.9961
Validation loss: 0.3245, Validation accuracy: 0.9236
```

```
Current learning rate is 0.001000
Epoch 168, iteration 59136:
Training loss: 0.0136, Training accuracy: 0.9966
Validation loss: 0.3261, Validation accuracy: 0.9234

Current learning rate is 0.001000
Epoch 169, iteration 59488:
Training loss: 0.0137, Training accuracy: 0.9964
Validation loss: 0.3295, Validation accuracy: 0.9234

Current learning rate is 0.001000
Epoch 170, iteration 59840:
Training loss: 0.0143, Training accuracy: 0.9960
Validation loss: 0.3267, Validation accuracy: 0.9228

Current learning rate is 0.001000
Epoch 171, iteration 60192:
Training loss: 0.0137, Training accuracy: 0.9964
Validation loss: 0.3258, Validation accuracy: 0.9246

Current learning rate is 0.001000
Epoch 172, iteration 60544:
Training loss: 0.0136, Training accuracy: 0.9965
Validation loss: 0.3266, Validation accuracy: 0.9262
Saving …

Current learning rate is 0.001000
Epoch 173, iteration 60896:
Training loss: 0.0139, Training accuracy: 0.9966
Validation loss: 0.3246, Validation accuracy: 0.9248

Current learning rate is 0.001000
Epoch 174, iteration 61248:
Training loss: 0.0142, Training accuracy: 0.9963
Validation loss: 0.3244, Validation accuracy: 0.9232

Current learning rate is 0.001000
Epoch 175, iteration 61600:
Training loss: 0.0139, Training accuracy: 0.9963
Validation loss: 0.3284, Validation accuracy: 0.9244

Current learning rate is 0.001000
Epoch 176, iteration 61952:
Training loss: 0.0130, Training accuracy: 0.9967
Validation loss: 0.3248, Validation accuracy: 0.9236

Current learning rate is 0.001000
```

```
Epoch 177, iteration 62304:
Training loss: 0.0134, Training accuracy: 0.9966
Validation loss: 0.3245, Validation accuracy: 0.9244

Current learning rate is 0.001000
Epoch 178, iteration 62656:
Training loss: 0.0125, Training accuracy: 0.9969
Validation loss: 0.3213, Validation accuracy: 0.9256

Current learning rate is 0.001000
Epoch 179, iteration 63008:
Training loss: 0.0131, Training accuracy: 0.9968
Validation loss: 0.3227, Validation accuracy: 0.9232

Current learning rate is 0.001000
Epoch 180, iteration 63360:
Training loss: 0.0145, Training accuracy: 0.9962
Validation loss: 0.3243, Validation accuracy: 0.9234

Current learning rate is 0.001000
Epoch 181, iteration 63712:
Training loss: 0.0125, Training accuracy: 0.9971
Validation loss: 0.3239, Validation accuracy: 0.9238

Current learning rate is 0.001000
According to ResNet paper, should terminate
Epoch 182, iteration 64064:
Training loss: 0.0132, Training accuracy: 0.9967
Validation loss: 0.3255, Validation accuracy: 0.9244

Current learning rate is 0.001000
According to ResNet paper, should terminate
Epoch 183, iteration 64416:
Training loss: 0.0135, Training accuracy: 0.9967
Validation loss: 0.3251, Validation accuracy: 0.9232

Current learning rate is 0.001000
According to ResNet paper, should terminate
Epoch 184, iteration 64768:
Training loss: 0.0129, Training accuracy: 0.9970
Validation loss: 0.3238, Validation accuracy: 0.9256

Current learning rate is 0.001000
According to ResNet paper, should terminate
Epoch 185, iteration 65120:
Training loss: 0.0134, Training accuracy: 0.9967
Validation loss: 0.3285, Validation accuracy: 0.9242
```

```
Current learning rate is 0.001000
According to ResNet paper, should terminate
Epoch 186, iteration 65472:
Training loss: 0.0128, Training accuracy: 0.9967
Validation loss: 0.3256, Validation accuracy: 0.9236

Current learning rate is 0.001000
According to ResNet paper, should terminate
Epoch 187, iteration 65824:
Training loss: 0.0124, Training accuracy: 0.9968
Validation loss: 0.3255, Validation accuracy: 0.9242

Current learning rate is 0.001000
According to ResNet paper, should terminate
Epoch 188, iteration 66176:
Training loss: 0.0129, Training accuracy: 0.9966
Validation loss: 0.3293, Validation accuracy: 0.9238

Current learning rate is 0.001000
According to ResNet paper, should terminate
Epoch 189, iteration 66528:
Training loss: 0.0125, Training accuracy: 0.9968
Validation loss: 0.3251, Validation accuracy: 0.9240

Current learning rate is 0.001000
According to ResNet paper, should terminate
Epoch 190, iteration 66880:
Training loss: 0.0123, Training accuracy: 0.9974
Validation loss: 0.3264, Validation accuracy: 0.9238

Current learning rate is 0.001000
According to ResNet paper, should terminate
Epoch 191, iteration 67232:
Training loss: 0.0129, Training accuracy: 0.9970
Validation loss: 0.3271, Validation accuracy: 0.9230

Current learning rate is 0.001000
According to ResNet paper, should terminate
Epoch 192, iteration 67584:
Training loss: 0.0124, Training accuracy: 0.9970
Validation loss: 0.3276, Validation accuracy: 0.9250

Current learning rate is 0.001000
According to ResNet paper, should terminate
Epoch 193, iteration 67936:
Training loss: 0.0128, Training accuracy: 0.9968
Validation loss: 0.3283, Validation accuracy: 0.9242
```

```
Current learning rate is 0.001000
According to ResNet paper, should terminate
Epoch 194, iteration 68288:
Training loss: 0.0135, Training accuracy: 0.9966
Validation loss: 0.3266, Validation accuracy: 0.9224

Current learning rate is 0.001000
According to ResNet paper, should terminate
Epoch 195, iteration 68640:
Training loss: 0.0129, Training accuracy: 0.9968
Validation loss: 0.3264, Validation accuracy: 0.9246

Current learning rate is 0.001000
According to ResNet paper, should terminate
Epoch 196, iteration 68992:
Training loss: 0.0133, Training accuracy: 0.9965
Validation loss: 0.3283, Validation accuracy: 0.9230

Current learning rate is 0.001000
According to ResNet paper, should terminate
Epoch 197, iteration 69344:
Training loss: 0.0124, Training accuracy: 0.9970
Validation loss: 0.3274, Validation accuracy: 0.9240

Current learning rate is 0.001000
According to ResNet paper, should terminate
Epoch 198, iteration 69696:
Training loss: 0.0123, Training accuracy: 0.9971
Validation loss: 0.3286, Validation accuracy: 0.9230

Current learning rate is 0.001000
According to ResNet paper, should terminate
Epoch 199, iteration 70048:
Training loss: 0.0121, Training accuracy: 0.9969
Validation loss: 0.3340, Validation accuracy: 0.9248

================================================
==> Optimization finished! Best validation accuracy: 0.9262
```

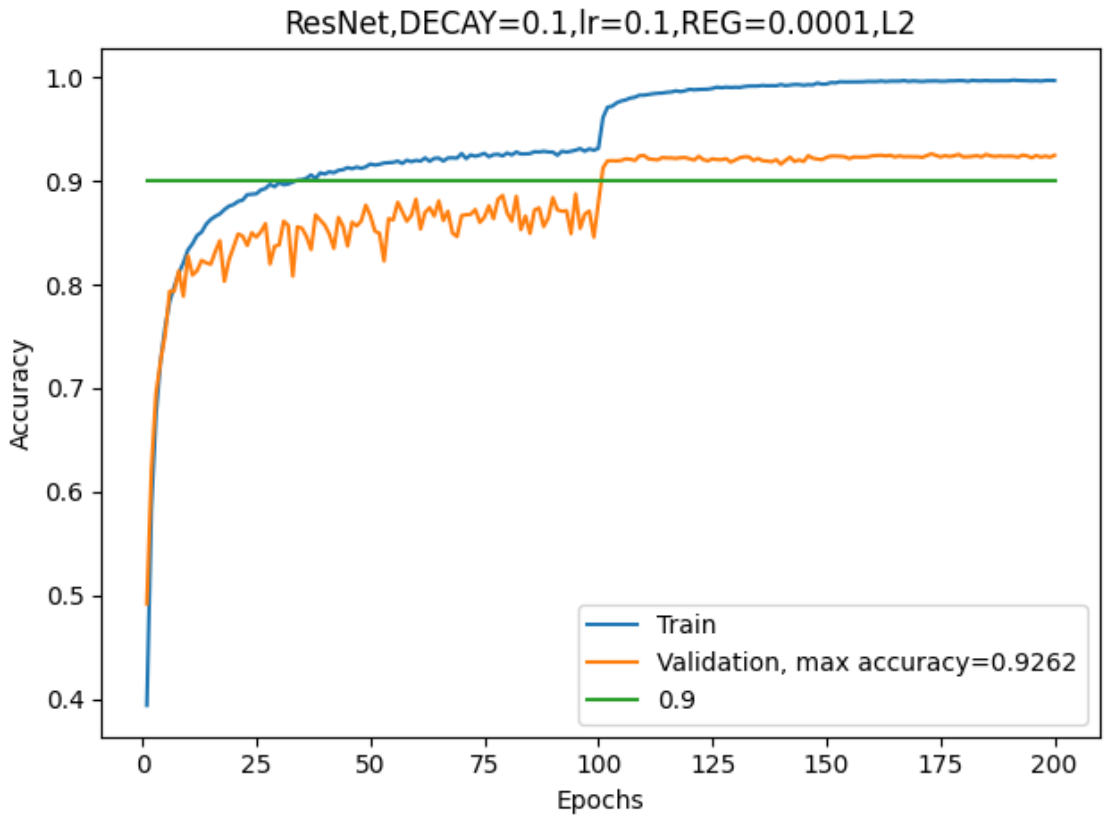
2 Bonus: with learning rate decay

The following code can help you adjust the learning rate during training. You need to figure out how to incorporate this code into your training loop.

```python
    if i % DECAY_EPOCHS == 0 and i != 0:
        current_learning_rate = current_learning_rate * DECAY
        for param_group in optimizer.param_groups:
            param_group['lr'] = current_learning_rate
```

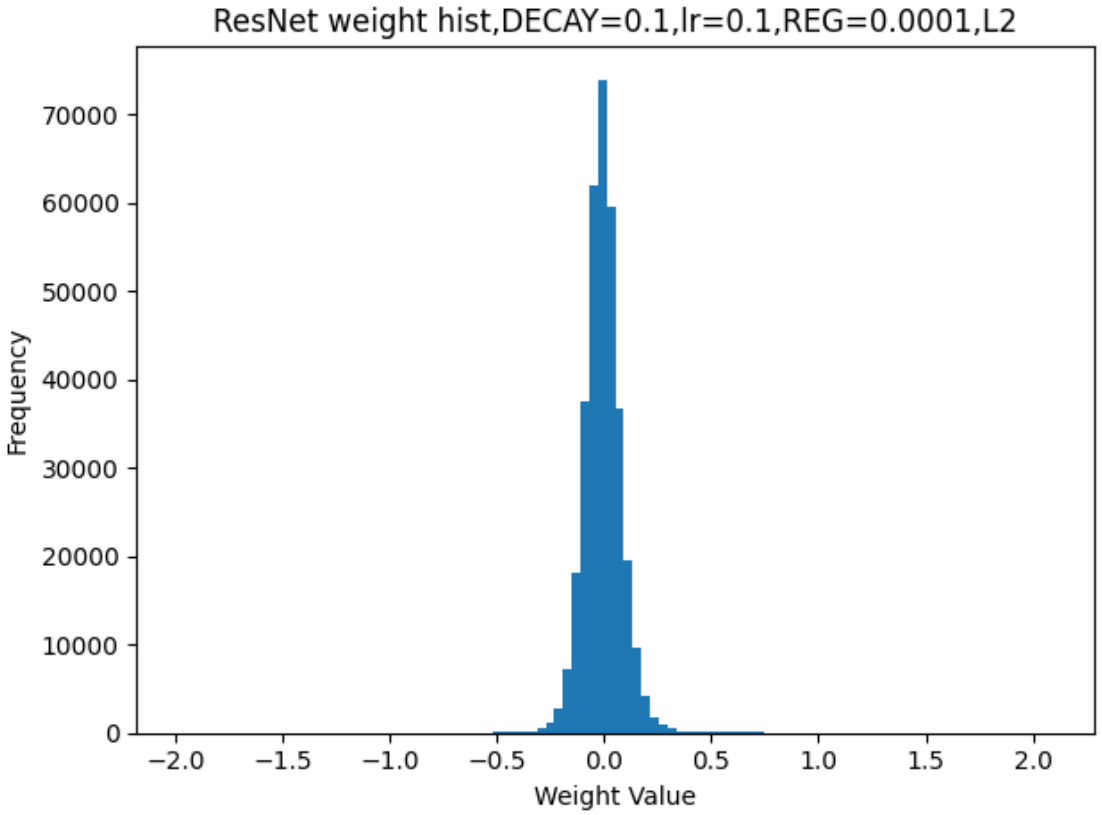
```
        print("Current learning rate has decayed to %f" %current_learning_rate)
```

[21]:
```
##### plotting accuracy vs epoch


import matplotlib.pyplot as plt
import numpy as np


def my_to_np(lst):
    return np.array([lst[i].detach().cpu().numpy().flatten() for i in␣
 ↪range(len(lst))])


save = 'y'

fig, ax = plt.subplots(1, 1)
xx = np.linspace(1, EPOCHS, EPOCHS)
## don't call my_to_np(lst) if plotting raises an error
ax.plot(xx, my_to_np(accuracy_arr_train), label='Train')
ax.plot(xx, my_to_np(accuracy_arr_val), label='Validation, max accuracy={:.4f}'.
 ↪format(best_val_acc))
ax.plot(xx, np.linspace(0.9, 0.9, EPOCHS), label='0.9')
title_ = 'ResNet,DECAY={:g},lr={:g},REG={:g},{:s}'.format(DECAY, INITIAL_LR,␣
 ↪REG, L1_or_L2)
if DECAY == 1:
    title_ = 'No Learning Rate Decay'
ax.set_xlabel('Epochs')
ax.set_ylabel('Accuracy')
ax.legend()
ax.set_title(title_)
fig.tight_layout()
if save == 'y':
    plt.savefig('q3_aug_BN_ReLU_lr{:g}_{:s}_{:g}.pdf'.format(INITIAL_LR,␣
 ↪L1_or_L2, REG),
                dpi=500, bbox_inches='tight')
```



[23]:
```
##### plotting weights distribution

save = 'y'
fig, ax = plt.subplots(1, 1)

weights_list = []

for name, module in model.named_modules():
    if isinstance(module, nn.Conv2d) or isinstance(module, nn.Linear):
        weights_list.extend(list((module.weight.cpu().detach().numpy()).
 ↪flatten()))

ax.hist(weights_list, bins=100)
ax.set_title(str("ResNet weight hist,DECAY={:g},lr={:g},REG={:g},{:s}".
 ↪format(DECAY, INITIAL_LR, REG, L1_or_L2)))

ax.set_xlabel('Weight Value')
ax.set_ylabel('Frequency')
fig.tight_layout()
if save == 'y':
```

```
    plt.savefig('q3weights_aug_BN_ReLU_lr{:g}_{:s}_{:g}.pdf'.format(INITIAL_LR,␣
↪L1_or_L2, REG),
                dpi=500, bbox_inches='tight')
```



[ ]: