Homework #3

CSE 446/546: Machine Learning
Profs. Simon Du and Sewoong Oh
Due: **Wednesday** May 19, 2021 11:59pm Pacific Time

Please review all homework guidance posted on the website before submitting to Gradescope. Reminders:

- Make sure to read the "What to Submit" section following each question and include all items.
- Please provide succinct answers and supporting reasoning for each question. Similarly, when discussing experimental results, concisely create tables and/or figures when appropriate to organize the experimental results. All explanations, tables, and figures for any particular part of a question must be grouped together.
- For every problem involving generating plots, please include the plots as part of your PDF submission.
- When submitting to Gradescope, please link each question from the homework in Gradescope to the location of its answer in your homework PDF. Failure to do so may result in deductions of up to *[5 points]* . For instructions, see https://www.gradescope.com/get_started#student-submission.
- Please recall that B problems, indicated in $\boxed{\text{boxed text}}$, are only graded for 546 students, and that they will be weighted at most 0.2 of your final GPA (see website for details). In Gradescope there is a place to submit solutions to A and B problems separately. You are welcome to create just a single PDF that contains answers to both, submit the same PDF twice, but associate the answers with the individual questions in Gradescope.
- If you collaborate on this homework with others, you must indicate who you worked with on your homework. Failure to do so may result in accusations of plagiarism.
- For every problem involving code, please include the code as part of your PDF for the PDF submission *in addition to* submitting your code to the separate assignment on Gradescope created for code. Not submitting all code files will lead to a deduction of *[1 point]* .

Not adhering to these reminders may result in point deductions.

Conceptual Questions

A1. The answers to these questions should be answerable without referring to external materials. Briefly justify your answers with a few words.

a. *[2 points]* Say you trained an SVM classifier with an RBF kernel ($K(u,v) = \exp\left(-\frac{\|u-v\|_2^2}{2\sigma^2}\right)$). It seems to underfit the training set: should you increase or decrease σ?

b. *[2 points]* True or False: Training deep neural networks requires minimizing a non-convex loss function, and therefore gradient descent might not reach the globally-optimal solution.

c. *[2 points]* True or False: It is a good practice to initialize all weights to zero when training a deep neural network.

d. *[2 points]* True or False: We use non-linear activation functions in a neural network's hidden layers so that the network learns non-linear decision boundaries.

e. *[2 points]* True or False: Given a neural network, the time complexity of the backward pass step in the backpropagation algorithm can be prohibitively larger compared to the relatively low time complexity of the forward pass step.

What to Submit:

- For each part a-e, 1-2 sentences containing your answer.

Kernels and the Bootstrap

A2. *[5 points]* Suppose that our inputs x are one-dimensional and that our feature map is infinite-dimensional: $\phi(x)$ is a vector whose ith component is

$$\frac{1}{\sqrt{i!}}\mathrm{e}^{-x^2/2}x^i \ ,$$

for all nonnegative integers i. (Thus, ϕ is an infinite-dimensional vector.) Show that $K(x,x') = \mathrm{e}^{-\frac{(x-x')^2}{2}}$ is a kernel function for this feature map, i.e.,

$$\phi(x)\cdot\phi(x') = \mathrm{e}^{-\frac{(x-x')^2}{2}} \ .$$

Hint: Use the Taylor expansion of $z \mapsto \mathrm{e}^z$. (This is the one dimensional version of the Gaussian (RBF) kernel).

What to Submit:

- A proof.

A3. This problem will get you familiar with kernel ridge regression using the polynomial and RBF kernels. First, let's generate some data. Let $n = 30$ and $f_*(x) = 4\sin(\pi x)\cos(6\pi x^2)$. For $i = 1,\ldots,n$ let each x_i be drawn uniformly at random from $[0,1]$, and let $y_i = f_*(x_i) + \epsilon_i$ where $\epsilon_i \sim \mathcal{N}(0,1)$.
For any function f, the true error and the train error are respectively defined as

$$\mathcal{E}_{\text{true}}(f) = \mathbb{E}_{X,Y}[(f(X)-Y)^2], \qquad \widehat{\mathcal{E}}_{\text{train}}(f) = \frac{1}{n}\sum_{i=1}^{n}\left(f(x_i)-y_i\right)^2 .$$

Using kernel ridge regression, construct a predictor

$$\widehat{\alpha} = \arg\min_{\alpha} \|K\alpha - y\|_2^2 + \lambda\alpha^\top K\alpha \ , \qquad \widehat{f}(x) = \sum_{i=1}^{n}\widehat{\alpha}_i k(x_i,x)$$

where $K \in \mathbb{R}^{n\times n}$ is the kernel matrix such that $K_{i,j} = k(x_i, x_j)$, and $\lambda \geq 0$ is the regularization constant. Include any code you use for your experiments in your submission.

a. *[10 points]* Using leave-one-out cross validation, find a good λ and hyperparameter settings for the following kernels:

   - $k_{\text{poly}}(x, z) = (1 + x^\top z)^d$ where $d \in \mathbb{N}$ is a hyperparameter,
   - $k_{\text{rbf}}(x, z) = \exp(-\gamma\|x - z\|_2^2)$ where $\gamma > 0$ is a hyperparameter[1].

   Report the values of d, γ, and the λ values for both kernels.

b. *[10 points]* Let $\widehat{f}_{\text{poly}}(x)$ and $\widehat{f}_{\text{rbf}}(x)$ be the functions learned using the hyperparameters you found in part a. For a single plot per function $\widehat{f} \in \left\{\widehat{f}_{\text{poly}}(x), \widehat{f}_{\text{rbf}}(x)\right\}$, plot the original data $\{(x_i, y_i)\}_{i=1}^n$, the true $f(x)$, and $\widehat{f}(x)$ (i.e., define a fine grid on $[0, 1]$ to plot the functions).

c. *[5 points]* We wish to build bootstrap percentile confidence intervals for $\widehat{f}_{\text{poly}}(x)$ and $\widehat{f}_{\text{rbf}}(x)$ for all $x \in [0, 1]$ from part b.[2] Use the non-parametric bootstrap with $B = 300$ bootstrap iterations to find 5% and 95% percentiles at each point x on a fine grid over $[0, 1]$.

   Specifically, for each bootstrap sample $b \in \{1, \dots, B\}$, draw uniformly at randomly with replacement, n samples from $\{(x_i, y_i)\}_{i=1}^n$, train an $\widehat{f}_b$ using the b-th resampled dataset, compute $\widehat{f}_b(x)$ for each x in your fine grid; let the 5% percentile at point x be the largest value ν such that $\frac{1}{B}\sum_{b=1}^{B} \mathbf{1}\{\widehat{f}_b(x) \leq \nu\} \leq .05$, define the 95% percentile analogously.

   Plot the 5 and 95 percentile curves on the plots from part b.

d. *[5 points]* Repeat parts a, b, and c with $n = 300$, but use 10-fold CV instead of leave-one-out for part a.

e. *[5 points]* For this problem, use the $\widehat{f}_{\text{poly}}(x)$ and $\widehat{f}_{\text{rbf}}(x)$ learned in part d. Suppose $m = 1000$ additional samples $(x'_1, y'_1), \dots, (x'_m, y'_m)$ are drawn i.i.d. the same way the first n samples were drawn.

   Use the non-parametric bootstrap with $B = 300$ to construct a confidence interval on

$$\mathbb{E}\left[\left(Y - \widehat{f}_{\text{poly}}(X)\right)^2 - \left(Y - \widehat{f}_{\text{rbf}}(X)\right)^2\right]$$

   (i.e. randomly draw with replacement m samples denoted as $\{(\widetilde{x}'_i, \widetilde{y}'_i)\}_{i=1}^m$ from $\{(x'_i, y'_i)\}_{i=1}^m$ and compute $\frac{1}{m}\sum_{i=1}^{m}\left[\left(\widetilde{y}'_i - \widehat{f}_{\text{poly}}(\widetilde{x}'_i)\right)^2 - \left(\widetilde{y}'_i - \widehat{f}_{\text{rbf}}(\widetilde{x}'_i)\right)^2\right]$, repeat this B times) and find 5% and 95% percentiles. Report these values.

   Using this confidence interval, is there statistically significant evidence to suggest that one of $\widehat{f}_{\text{rbf}}$ and $\widehat{f}_{\text{poly}}$ is better than the other at predicting Y from X? (Hint: does the confidence interval contain 0?)

What to Submit:

- Report the values of d, γ and the value of λ for both kernels as described in part (a).
- The two separate plots described in part (b).
- The two separate plots described in part (c) - part (b)'s plots with the 5 and 95 percentile curves.
- For part (d) provide the values of d, γ, and the value of λ for both kernels. In addition, provide two separate plots as you did for part (b) then two more plots as you did for part (c).

[1]Given a dataset $x_1, \dots, x_n \in \mathbb{R}^d$, a heuristic for choosing a range of γ in the right ballpark is the inverse of the median of all $\binom{n}{2}$ squared distances $\|x_i - x_j\|_2^2$.

[2]See Hastie, Tibshirani, Friedman Ch. 8.2 for a review of the bootstrap procedure.

- Report the 5% and 95% percentiles for part (e). In addition, in 1-2 sentences, comment on if there is statistically significant evidence to suggest that one kernel performs better than the other.
- Your code implementation for each part.

B1.

Intro to sample complexity

For $i = 1, \dots, n$ let $(x_i, y_i) \overset{\text{i.i.d.}}{\sim} P_{X,Y}$ where $y_i \in \{-1, 1\}$ and x_i lives in some set $\mathcal{X}$ (x_i is not necessarily a vector). The 0/1 loss, or *risk*, for a deterministic classifier $f\colon \mathcal{X} \to \{-1, 1\}$ is defined as:

$$R(f) = \mathbb{E}_{X,Y}[\mathbf{1}(f(X) \neq Y)]$$

where $\mathbf{1}(\mathcal{E})$ is the indicator function for the event $\mathcal{E}$ (the function takes the value 1 if $\mathcal{E}$ occurs and 0 otherwise). The expectation is with respect to the underlying distribution $P_{X,Y}$ on (X, Y). Unfortunately, we don't know $P_{X,Y}$ exactly, but we do have our i.i.d. samples $\{(x_i, y_i)\}_{i=1}^n$ drawn from it. Define the *empirical risk* as

$$\widehat{R}_n(f) = \frac{1}{n} \sum_{i=1}^{n} \mathbf{1}(f(x_i) \neq y_i) \ ,$$

which is just an empirical estimate of our risk. Suppose that a learning algorithm computes the empirical risk $R_n(f)$ for all $f \in \mathcal{F}$ and outputs the prediction function $\widehat{f}$ which is the one with the smallest empirical risk. (In this problem, we are assuming that $\mathcal{F}$ is finite.) Suppose that the best-in-class function f^* (i.e., the one that minimizes the true 0/1 loss) is:

$$f^* = \arg\min_{f \in \mathcal{F}} R(f) \,.$$

a. *[2 points]* Suppose that for some $f \in \mathcal{F}$, we have $R(f) > \epsilon$. Show that

$$\mathbb{P}\left[\widehat{R}_n(f) = 0\right] \leq \mathrm{e}^{-n\epsilon} \ .$$

(You may use the fact that $1 - \epsilon \leq \mathrm{e}^{-\epsilon}$.)

b. *[2 points]* Use the *union bound* to show that

$$\mathbb{P}\left[\exists f \in \mathcal{F} \text{ s.t. } R(f) > \epsilon \text{ and } \widehat{R}_n(f) = 0\right] \leq |\mathcal{F}|\mathrm{e}^{-\epsilon n}.$$

Recall that the union bound says that if $A_1, \dots, A_k$ are events in a probability space, then

$$\mathbb{P}\left[A_1 \cup A_2 \cup \ldots \cup A_k\right] \leq \sum_{1 \leq i \leq k} \mathbb{P}(A_i).$$

c. *[2 points]* Solve for the minimum ϵ such that $|\mathcal{F}|\mathrm{e}^{-\epsilon n} \leq \delta$.

d. *[4 points]* Use this to show that with probability at least $1 - \delta$

$$\widehat{R}_n(\widehat{f}) = 0 \quad \Longrightarrow \quad R(\widehat{f}) - R(f^*) \leq \frac{\log(|\mathcal{F}|/\delta)}{n}$$

where $\widehat{f} = \arg\min_{f \in \mathcal{F}} \widehat{R}_n(f)$.

Context: Note that among a larger number of functions $\mathcal{F}$ there is more likely to exist an $\widehat{f}$ such that $\widehat{R}_n(\widehat{f}) = 0$. However, this increased flexibility comes at the cost of a worse guarantee on the true error reflected in the larger $|\mathcal{F}|$. This trade-off quantifies how we can choose function classes $\mathcal{F}$ that over fit. This sample complexity result is remarkable because it depends just on the number of functions in $\mathcal{F}$, not what they look like. This is among the simplest results among a rich literature known as 'Statistical Learning Theory'. Using a similar strategy, one can use Hoeffding's inequality to obtain a generalization bound when $\widehat{R}_n(\widehat{f}) \neq 0$.

What to Submit:

- A proof that $\mathbb{P}\left[\widehat{R}_n(f) = 0\right] \leq \mathrm{e}^{-n\epsilon}$ for part (a).
- A proof that $\mathbb{P}\left[\exists f \in \mathcal{F} \text{ s.t. } R(f) > \epsilon \text{ and } \widehat{R}_n(f) = 0\right] \leq |\mathcal{F}|\mathrm{e}^{-\epsilon n}$ for part (b).
- A solution finding the minimum that satisfies the equation in part (c).
- A proof that $\widehat{R}_n(\widehat{f}) = 0 \implies R(\widehat{f}) - R(f^*) \leq \frac{\log(|\mathcal{F}|/\delta)}{n}$ for part (d).

B2.

Perceptron

One of the oldest algorithms used in machine learning (from early 60's) is an online algorithm for learning a linear threshold function called the Perceptron Algorithm:

1. Start with the all-zeroes weight vector $\mathbf{w}_1 = 0$, and initialize t to 1. Also let's automatically scale all examples $\mathbf{x}$ to have (Euclidean) norm 1, since this doesn't affect which side of the plane they are on.
2. Given example $\mathbf{x}$, predict positive iff $\mathbf{w}_t \cdot \mathbf{x} > 0$.
3. On a mistake, update as follows:
   - Mistake on positive: $\mathbf{w}_{t+1} \leftarrow \mathbf{w}_t + \mathbf{x}$.
   - Mistake on negative: $\mathbf{w}_{t+1} \leftarrow \mathbf{w}_t - \mathbf{x}$.

   $t \leftarrow t + 1$.

If we make a mistake on a positive $\mathbf{x}$ we get $\mathbf{w}_{t+1} \cdot \mathbf{x} = (\mathbf{w}_t + \mathbf{x}) \cdot \mathbf{x} = \mathbf{w}_t \cdot \mathbf{x} + 1$, and similarly if we make a mistake on a negative $\mathbf{x}$ we have $\mathbf{w}_{t+1} \cdot \mathbf{x} = (\mathbf{w}_t - \mathbf{x}) \cdot \mathbf{x} = \mathbf{w}_t \cdot \mathbf{x} - 1$. So, in both cases we move closer (by 1) to the value we wanted. Here is a link if you are interested in more details.

Now consider the linear decision boundary for classification (labels in $\{-1, 1\}$) of the form $\mathbf{w} \cdot \mathbf{x} = 0$ (i.e., no offset). Now consider the following loss function evaluated at a data point $(\mathbf{x}, y)$ which is a variant on the hinge loss.

$$\ell((\mathbf{x}, y), \mathbf{w}) = \max\{0, -y(\mathbf{w} \cdot \mathbf{x})\}.$$

a. *[2 points]* Given a dataset of $(\mathbf{x}_i, y_i)$ pairs, write down a single step of subgradient descent with a step size of η if we are trying to minimize

$$\frac{1}{n} \sum_{i=1}^{n} \ell((\mathbf{x}_i, y_i), \mathbf{w})$$

for $\ell(\cdot)$ defined as above. That is, given a current iterate $\widetilde{\mathbf{w}}$ what is an expression for the next iterate?

b. *[2 points]* Use what you derived to argue that the Perceptron can be viewed as implementing SGD applied to the loss function just described (for what value of η)?

c. *[1 point]* Suppose your data was drawn i.i.d. and that there exists a $\mathbf{w}^*$ that separates the two classes perfectly. Provide an explanation for why hinge loss is generally preferred over the loss given above.

What to Submit:

- For (a) a single step of subgradient descent
- An answer to the question proposed in part (b).
- An explanation as described in part (c).

Neural Networks for MNIST

Resources

For questions A.4, A.5 and A.6 you will use a lot of PyTorch. In Section materials (Week 6) there is a notebook that you might find useful. Additionally make use of PyTorch Documentation, when needed.
If you do not have access to GPU, you might find Google Colaboratory useful. It allows you to use a cloud GPU for free. To enable it make sure: "Runtime" -¿ "Change runtime type" -¿ "Hardware accelerator" is set to "GPU". When submitting please download and submit a `.py` version of your notebook.

A4. In Homework 1, we used ridge regression for training a classifier for the MNIST data set. In Homework 2, we used logistic regression to distinguish between the digits 2 and 7. In this problem, we will use PyTorch to build a simple neural network classifier for MNIST to further improve our accuracy.

We will implement two different architectures: a shallow but wide network, and a narrow but deeper network. For both architectures, we use d to refer to the number of input features (in MNIST, $d = 28^2 = 784$), h_i to refer to the dimension of the i-th hidden layer and k for the number of target classes (in MNIST, $k = 10$). For the non-linear activation, use ReLU. Recall from lecture that

$$\text{ReLU}(x) = \begin{cases} x, & x \geq 0 \\ 0, & x < 0 \end{cases}.$$

Weight Initialization

Consider a weight matrix $W \in \mathbb{R}^{n\times m}$ and $b \in \mathbb{R}^n$. Note that here m refers to the input dimension and n to the output dimension of the transformation $x \mapsto Wx + b$. Define $\alpha = \frac{1}{\sqrt{m}}$. Initialize all your weight matrices and biases according to $\text{Unif}(-\alpha, \alpha)$.

Training

For this assignment, use the Adam optimizer from `torch.optim`. Adam is a more advanced form of gradient descent that combines momentum and learning rate scaling. It often converges faster than regular gradient descent in practice. You can use either Gradient Descent or any form of Stochastic Gradient Descent. Note that you are still using Adam, but might pass either the full data, a single datapoint or a batch of data to it. Use cross entropy for the loss function and ReLU for the non-linearity.

Implementing the Neural Networks

a. *[10 points]* Let $W_0 \in \mathbb{R}^{h\times d}$, $b_0 \in \mathbb{R}^h$, $W_1 \in \mathbb{R}^{k\times h}$, $b_1 \in \mathbb{R}^k$ and $\sigma(z)\colon \mathbb{R} \to \mathbb{R}$ some non-linear activation function applied element-wise. Given some $x \in \mathbb{R}^d$, the forward pass of the wide, shallow network can be formulated as:

$$\mathcal{F}_1(x) := W_1\sigma(W_0x + b_0) + b_1$$

Use $h = 64$ for the number of hidden units and choose an appropriate learning rate. Train the network until it reaches 99% accuracy on the training data and provide a training plot (loss vs. epoch). Finally evaluate the model on the test data and report both the accuracy and the loss.

b. *[10 points]* Let $W_0 \in \mathbb{R}^{h_0\times d}$, $b_0 \in \mathbb{R}^{h_0}$, $W_1 \in \mathbb{R}^{h_1\times h_0}$, $b_1 \in \mathbb{R}^{h_1}$, $W_2 \in \mathbb{R}^{k\times h_1}$, $b_2 \in \mathbb{R}^{k}$ and $\sigma(z) : \mathbb{R} \to \mathbb{R}$ some non-linear activation function. Given some $x \in \mathbb{R}^d$, the forward pass of the network can be formulated as:

$$\mathcal{F}_2(x) := W_2\sigma(W_1\sigma(W_0x + b_0) + b_1) + b_2$$

Use $h_0 = h_1 = 32$ and perform the same steps as in part a.

c. *[5 points]* Compute the total number of parameters of each network and report them. Then compare the number of parameters as well as the test accuracies the networks achieved. Is one of the approaches (wide, shallow vs. narrow, deeper) better than the other? Give an intuition for why or why not.

Using PyTorch: For your solution, you may not use any functionality from the `torch.nn` module except for `torch.nn.functional.relu` and `torch.nn.functional.cross_entropy`. You must implement the networks $\mathcal{F}_1$ and $\mathcal{F}_2$ from scratch. For starter code and a tutorial on PyTorch refer to the sections 6 and 7 material.

What to Submit:

- Provide a plot of the training loss versus epoch for part (a). In addition evaluate the model trained in part (a) on the test data and report the accuracy and loss.
- Provide a plot of the training loss versus epoch for part (b). In addition evaluate the model trained in part (b) on the test data and report the accuracy and loss.
- Report the number of parameters for the network trained in part (a) and for the network trained in part (b). Provide a comparison of the two networks as described in part (c).
- Your code implementing each part.

Using Pretrained Networks and Transfer Learning

A5. So far we have trained very small neural networks from scratch. As mentioned in the previous problem, modern neural networks are much larger and more difficult to train and validate. In practice, it is rare to train such large networks from scratch. This is because it is difficult to obtain both the massive datasets and the computational resources required to train such networks.

Instead of training a network from scratch, in this problem, we will use a network that has already been trained on a very large dataset (ImageNet) and adjust it for the task at hand. This process of adapting weights in a model trained for another task is known as *transfer learning*.

- Begin with the pretrained AlexNet model from `torchvision.models` for both tasks below. AlexNet achieved an early breakthrough performance on `ImageNet` and was instrumental in sparking the deep learning revolution in 2012.
- Do not modify any module within AlexNet that is not the final classifier layer.
- The output of AlexNet comes from the 6-th layer of the classifier. Specifically, `model.classifer[6] = nn.Linear(4096, 1000)`. To use AlexNet with CIFAR-10, we will reinitialize (replace) this layer with `nn.Linear(4096, 10)`. This re-initializes the weights, and changes the output shape to reflect the desired number of target classes in CIFAR-10.

We will explore two different ways to formulate transfer learning.

a. *[15 points]* **Use AlexNet as a fixed feature extractor:** Add a new linear layer to replace the existing classification layer, and only adjust the weights of this new layer (keeping the weights of all other layers fixed). Provide plots for training loss and validation loss over the number of epochs. Report the highest validation accuracy achieved. Finally, evaluate the model on the test data and report both the accuracy and the loss.

   When using AlexNet as a fixed feature extractor, make sure to freeze all of the parameters in the network *before* adding your new linear layer:

```
model = torchvision.models.alexnet(pretrained=True)
for param in model.parameters():
    param.requires_grad = False
model.classifier[6] = nn.Linear(4096, 10)
```

b. *[15 points]* **Fine-Tuning:** The second approach to transfer learning is to fine-tune the weights of the pre-trained network, in addition to training the new classification layer. In this approach, all network weights are updated at every training iteration; we simply use the existing AlexNet weights as the "initialization" for our network (except for the weights in the new classification layer, which will be initialized using whichever method is specified in the constructor) prior to training on CIFAR-10. Following the same procedure, report all the same metrics and plots as in the previous question.

What to Submit:

- State the values of hyperparameters you chose to use for both parts (a) and (b).
- For part (a), provide a plot of the training loss and validation loss over the number of epochs (can be the same plot), report the highest validation accuracy achieved, and finally evaluate the model on the test set and report both the accuracy and the loss.
- For part (b), provide a plot of the training loss and validation loss over the number of epochs (can be the same plot), report the highest validation accuracy achieved, and finally evaluate the model on the test set and report both the accuracy and the loss.
- Your code implementing each part.

Image Classification on CIFAR-10

A6. In this problem we will explore different deep learning architectures for image classification on the CIFAR-10 dataset. If you are not comfortable with PyTorch from the previous lecture and discussion materials, use the tutorials at PyTorch's 60min blitz tutorial and make sure you are familiar with tensors, two-dimensional convolutions (`nn.Conv2d`) and fully-connected layers (`nn.Linear`), ReLU non-linearities (`F.relu`), pooling (`nn.MaxPool2d`), and tensor reshaping (`view`).

For this problem, it is highly recommended that you copy and modify the existing network code produced in the tutorial *Training a classifier*. You should not be coding this network from scratch!

A few preliminaries:

- Each network f maps an image $x^{\text{in}} \in \mathbb{R}^{32\times32\times3}$ (3 channels for RGB) to an output $f(x^{\text{in}}) = x^{\text{out}} \in \mathbb{R}^{10}$. The class label is predicted as $\arg\max_{i=0,1,\ldots,9} x_i^{\text{out}}$.
- The network is trained via multiclass cross-entropy loss.

- Create a validation dataset by appropriately partitioning the train dataset.
  Hint: look at the documentation for `torch.utils.data.random_split`. Make sure to tune hyperparameters like network architecture and step size on the validation dataset. Do **NOT** validate your hyperparameters on the test dataset.
- Modify the training code such that at the end of each epoch (one pass over the training data) it computes and prints the training and test classification accuracy.
- While one would usually train a network for hundreds of epochs to reach convergence and maximize accuracy, this can be prohibitively time-consuming, so feel free to train for just a dozen or so epochs.

For all of the following, apply a hyperparameter tuning method (grid search, random search, etc.) using the validation set, report the hyperparameter configurations you evaluated and the best set of hyperparameters from this set, and plot the training and validation classification accuracy as a function of iteration. Produce a separate line or plot for each hyperparameter configuration evaluated. Finally, evaluate your best set of hyperparameters on the test data and report the accuracy. **On the larger networks, you should expect to tune hyperparameters and train to at least 70% accuracy.**

Here are the network architectures you will construct and compare.

a. *[15 points]* **Fully-connected output, 0 hidden layers (logistic regression):** this network has no hidden layers and linearly maps the input layer to the output layer. This can be written as

$$x^{\text{out}} = W(x^{\text{in}}) + b$$

where $x^{\text{out}} \in \mathbb{R}^{10}$, $x^{\text{in}} \in \mathbb{R}^{32\times32\times3}$, $W \in \mathbb{R}^{10\times3072}$, $b \in \mathbb{R}^{10}$ since $3072 = 32\cdot32\cdot3$. For a tensor $x \in \mathbb{R}^{a\times b\times c}$, we let $(x) \in \mathbb{R}^{abc}$ be the reshaped form of the tensor into a vector (in an arbitrary but consistent pattern).

b. *[15 points]* **Fully-connected output, 1 fully-connected hidden layer:** this network has one hidden layer denoted as $x^{\text{hidden}} \in \mathbb{R}^M$ where M will be a hyperparameter you choose (M could be in the hundreds). The non-linearity applied to the hidden layer will be the `relu` ($\text{relu}(x) = \max\{0, x\}$. This network can be written as

$$x^{\text{out}} = W_2\text{relu}(W_1(x^{\text{in}}) + b_1) + b_2$$

where $W_1 \in \mathbb{R}^{M\times3072}$, $b_1 \in \mathbb{R}^M$, $W_2 \in \mathbb{R}^{10\times M}$, $b_2 \in \mathbb{R}^{10}$.

c. *[15 points]* **Convolutional layer with max-pool and fully-connected output:** for a convolutional layer W_1 with filters of size $k\times k\times 3$, and M filters (reasonable choices are $M = 100$, $k = 5$), we have that $\text{Conv2d}(x^{\text{in}}, W_1) \in \mathbb{R}^{(33-k)\times(33-k)\times M}$.

  - Each convolution will have its own offset applied to each of the output pixels of the convolution; we denote this as $\text{Conv2d}(x^{\text{in}}, W) + b_1$ where b_1 is parameterized in $\mathbb{R}^M$. Apply a `relu` activation to the result of the convolutional layer.
  - Next, use a max-pool of size $N\times N$ (a reasonable choice is $N = 14$ to pool to 2×2 with $k = 5$) we have that $\text{MaxPool}(\text{relu}(\text{Conv2d}(x^{\text{in}}, W_1) + b_1)) \in \mathbb{R}^{\lfloor\frac{33-k}{N}\rfloor\times\lfloor\frac{33-k}{N}\rfloor\times M}$.
  - We will then apply a fully-connected layer to the output to get a final network given as

  $$x^{\text{output}} = W_2(\text{MaxPool}(\text{relu}(\text{Conv2d}(x^{\text{input}}, W_1) + b_1))) + b_2$$

  where $W_2 \in \mathbb{R}^{10\times M(\lfloor\frac{33-k}{N}\rfloor)^2}$, $b_2 \in \mathbb{R}^{10}$.

  The parameters M, k, N (in addition to the step size and momentum) are all hyperparameters, but you can choose a reasonable value. Tuning can be performed (optionally) in the next subproblem.

d. *[5 points]* **Tuning:** Return to the original network you were left with at the end of the tutorial *Training a classifier*. Tune the different hyperparameters (number of convolutional filters, filter sizes, dimensionality of the fully-connected layers, stepsize, etc.) and train for a sufficient number of iterations to achieve a *test accuracy* of at least 70%. Provide the hyperparameter configuration used to achieve this performance.

The number of hyperparameters to tune in exercise (**??**), combined with the slow training times, will hopefully give you a taste of how difficult it is to construct networks with good generalization performance. State-of-the-art networks can have dozens of layers, each with their own hyperparameters to tune. Additional hyperparameters you are welcome to play with if you are so inclined, include: changing the activation function, replace max-pool with average-pool, and experimenting with batch normalization or dropout.

What to Submit:

- For parts (a, b, c, d) apply a hyperparameter tuning method of your choice using the validation set and report the best parameters you found. Plot the training and validation classification accuracy as a function of iteration. Produce a separate line or plot **for each hyperparameter configuration evaluated** (please try to use multiple lines in a single plot to keep the number of figures minimal), and make sure to state all hyperparameters and the grid of values you explored.
- For parts (a, b, c, d) provide a plot of the training and validation accuracy when training with the best hyperparameters (separate from your tuning plots).
- For parts (a, b, c, d) evaluate the the model trained with the best set of hyperparameters on the test data and report the train and test accuracy.
- Your code implementing each part.
- Note: Ensure that for the larger networks you train with your final hyperparameters to at least 70% *training* accuracy.