Assignment 2A

Introduction to Machine Learning: CS-3410-1 (Spring 2025)

April 16, 2025

Question 1: Kernels

1. Suppose that our inputs are one-dimensional and that our feature map is infinite-dimensional. The feature map $\phi(x)$ is defined such that the i-th component is:

$$\phi_i(x) = \frac{1}{\sqrt{i!}} e^{-x^2/2} x^i,$$

for all nonnegative integers i. Thus, $\phi(x)$ is an infinite-dimensional vector.

2. Show that the kernel function

$$K(x, x') = e^{-\frac{(x-x')^2}{2}}$$

is a valid kernel corresponding to the dot product of the feature maps:

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

Question 2: Derivation for Hard-margin Linear SVMs

1. Assume your data is linearly separable. What is the advantage of the hard-margin linear SVM solution over the solution found by the Perceptron (see section 2.2 in Andrew NG's notes)?

2. In class, we covered formulation A. Here are two equivalent formulations of the linear SVM, shown below:

Formulation A:

$$\begin{aligned} \min_{w,b} \quad & \|w\|^2 \\ \text{subject to} \quad & \forall i, \quad y_i\big(w^T x_i + b\big) \geq 1 \end{aligned}$$

Formulation B:

$$\begin{aligned} \min_{w,b} \quad & \|w\|^2 \\ \text{subject to} \quad & \forall i, \quad y_i\big(w^T x_i + b\big) \geq 0 \\ & \min_i \big| \, w^T x_i + b \, \big| = 1 \end{aligned}$$

3. Consider the following comparisons between Formulation A and Formulation B:

   (i) Assume you have found the optimal solution to the above optimization in A. Prove that the optimal solution of A is a feasible solution for B.

   (ii) Assume you have found the optimal solution to B. Prove that this solution is a feasible solution for A.

   (iii) Prove that the optimal solution in A is the optimal solution for B and vice versa.

Question 3: Hard vs. Soft margin SVMs

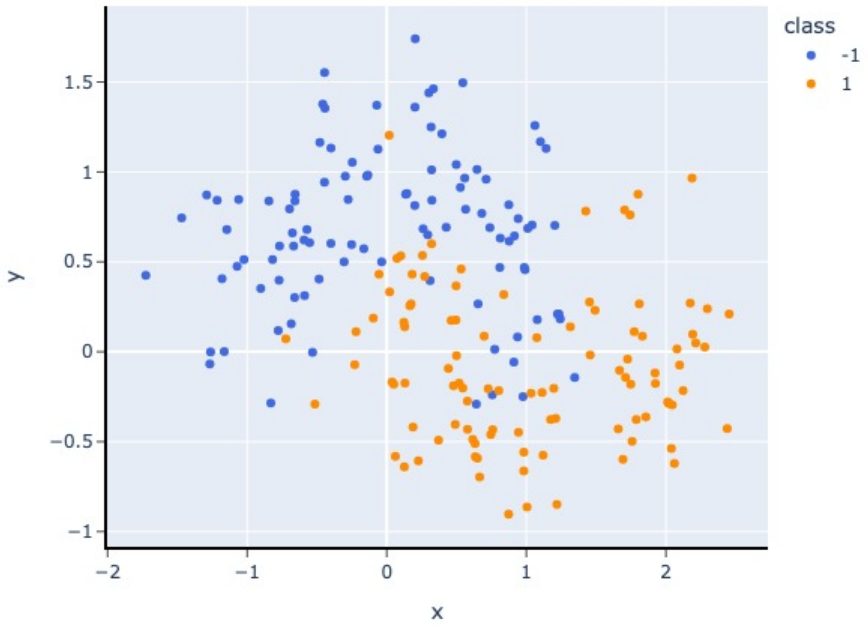
1. Does the hard-margin SVM converge for data that may not be linearly separable data (refer to the figure below)? How about the soft-margin SVM? Please explain in detail.

2. For the soft-margin linear SVM, we use the hyperparameter C to tune how much we penalize misclassifications. As $C \to \infty$, does the soft-margin SVM become more similar or less similar to the hard-margin SVM? As $C \to 0^+$, what happens to the solution of the soft-margin SVM? Why?

3. Suppose the data is linearly separable and you found a solution $(\mathbf{w}, b)$ to the hard-margin SVM. The operating hyperplane is surrounded by a margin defined by hyperplanes:

$$\{\mathbf{x} : \mathbf{w}^T\mathbf{x} + b = 1\} \quad \text{and} \quad \{\mathbf{x} : \mathbf{w}^T\mathbf{x} + b = -1\}.$$

   Prove that at least one training data point lies on each of these margin hyperplanes.



Question 4: Norm Soft Margin SVMs

If our data is not linearly separable, then we need to modify our approach by introducing an error margin that must be minimized. Specifically, we looked at SVM in the form known as the ℓ_1 norm soft margin SVM. In this problem, we will consider an alternative method, also known as the ℓ_2 norm soft margin SVM.

This new algorithm is given by the following optimization problem (notice that the slack penalties are now squared):

$$\begin{aligned} \min_{w,\,b,\,\xi} \quad & \frac{1}{2}\|w\|_2^2 \;+\; \frac{C}{2}\sum_{i=1}^{n}\xi_i^2 \\ \text{subject to} \quad & y_i\big(w^T x_i + b\big) \;\geq\; 1 \;-\; \xi_i, \quad i = 1, 2, \ldots, n \end{aligned}$$

NOTE: Let $\mathbf{x} = (x_1, x_2, \ldots, x_n) \in \mathbb{R}^n$ be a vector. Then:

- The ℓ_1-norm of $\mathbf{x}$ is defined as

$$\|\mathbf{x}\|_1 = \sum_{i=1}^{n} |x_i|.$$

- The ℓ_2-norm of $\mathbf{x}$ is defined as

$$\|\mathbf{x}\|_2 = \sqrt{\sum_{i=1}^{n} x_i^2}.$$

1. Notice that we have dropped the $\xi_i \geq 0$ constraint in the ℓ_2 problem. Show that these non-negativity constraints can be removed. In other words, prove that the optimal value of the objective will be the same whether or not these constraints are present.

2. Write down the Lagrangian of the ℓ_2 soft margin SVM optimization problem.

3. Minimize the Lagrangian with respect to w, b, and $\{\xi_i\}$ by taking the following gradients and setting them to zero:

$$\nabla_w \mathcal{L}, \quad \frac{\partial \mathcal{L}}{\partial b}, \quad \nabla_\xi \mathcal{L}.$$

Let $\xi = \begin{bmatrix}\xi_1, \xi_2, \ldots, \xi_n\end{bmatrix}^T$.

Question 5: Kernel Ridge Regression

1. In ridge regression, our cost function is regularized by adding $\lambda\|\theta\|^2$, where $\|\theta\|$ is the Euclidean norm (regularization term) and $\lambda > 0$ is a known constant, to the original cost function of linear regression. The ridge regression cost function is

$$J(\theta) = \sum_{i=1}^{n} \big(y_i - \theta^T x_i\big)^2 + \frac{\lambda}{2}\|\theta\|^2.$$

Use vector notation to represent a closed-form expression for the value of θ that minimizes the ridge regression cost function.

2. Suppose we want to use kernels to implicitly represent our feature vectors in a high-dimensional (possibly infinite-dimensional) space. Using a feature mapping ϕ, the ridge regression cost function becomes

$$J(\phi) = \sum_{i=1}^{n} \big(y_i - \theta^T \phi(x_i)\big)^2 + \frac{\lambda}{2}\|\theta\|^2.$$

Making a prediction on a new input $\mathbf{x}_{\text{new}}$ would now be done by computing the prediction

$$\hat{y}_{\text{new}} = \theta^T \phi(\mathbf{x}_{\text{new}}).$$

Show how we can use the "kernel trick" to obtain a closed-form expression for the prediction on the new input without ever explicitly computing $\phi(x_{\text{new}})$.

You may assume that the parameter vector θ can be expressed as a linear combination of the input feature vectors, i.e.,

$$\theta = \sum_{i=1}^{n} \alpha_i \, \phi(x_i)$$

for some set of parameters α_i.

Hint: You may find the following identity useful:

$$(\lambda I + BA)^{-1} B = B\big(\lambda I + AB\big)^{-1}.$$

Question 6: Simple Deep Learning Architecture

In this question, you are going to explore a 2-layer fully connected network with RELU activation function. Suppose you have the architecture above, namely,

$$\begin{bmatrix} h_1 \\ h_2 \end{bmatrix} = \begin{bmatrix} w_{11} & w_{12} & w_{13} \\ w_{21} & w_{22} & w_{23} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ 1 \end{bmatrix} \quad z = \begin{bmatrix} v_1 & v_2 & v_3 \end{bmatrix} \begin{bmatrix} f(h_1) \\ f(h_2) \\ 1 \end{bmatrix} \quad t = \sigma(z)$$

where $f(h_i) = \max(0, h_i)$, $\sigma(z) = \frac{1}{1+e^{-z}}$, and t is the output of the network.

1. Suppose

$$\begin{bmatrix} w_{11} & w_{12} & w_{13} \\ w_{21} & w_{22} & w_{23} \end{bmatrix} = \begin{bmatrix} 1 & -1 & 0 \\ -1 & -1 & 0 \end{bmatrix}, \quad \begin{bmatrix} v_1 & v_2 & v_3 \end{bmatrix} = \begin{bmatrix} 1 & 1 & -2 \end{bmatrix}$$

Draw the decision boundary of the network, namely, $\sigma(z) = 0.5$. Please indicate the positive and negative side of the boundary.

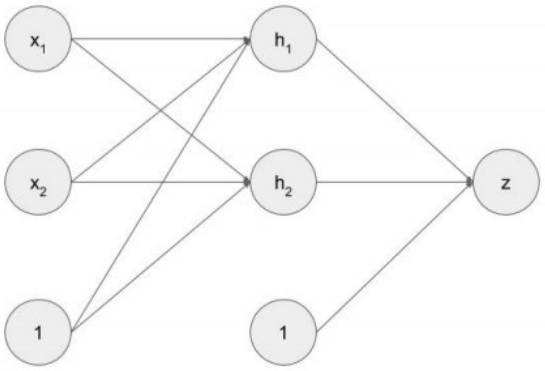


Figure 1: Figure illustrating the network.

2. Assume the weights in (a). What is the prediction for $[x_1, x_2]^T = [1, 1]^T$?

3. Usually, the weights

$$W = \begin{bmatrix} w_{11} & w_{12} & w_{13} \\ w_{21} & w_{22} & w_{23} \end{bmatrix}, \quad V = \begin{bmatrix} v_1 & v_2 & v_3 \end{bmatrix}$$

have to be learned using Stochastic Gradient Descent. For neural network binary classification, a common loss function is the cross-entropy loss:

$$l(y, t) = -(y \log(t) + (1 - y) \log(1 - t))$$

where y is the true label for a sample and t is the output of the neural network. In order to use SGD, we have to derive the gradient with respect to W and V. Show that for a single training example, $x = [x_1, x_2]$,

$$\frac{\partial l}{\partial v_i} = (t - y) f(h_i) \quad \text{for } i \neq 3$$

$$\frac{\partial l}{\partial v_3} = (t - y)$$

$$\frac{\partial l}{\partial w_{ij}} = (t - y) v_i \mathbf{1}(h_i > 0) x_j \quad \text{for } j \neq 3$$

$$\frac{\partial l}{\partial w_{i3}} = (t - y) v_i \mathbf{1}(h_i > 0)$$

where $\mathbf{1}(\cdot)$ is the indicator function.

Question 7: Deep Learning Architectures
In this problem we will explore different deep learning architectures for a classification task. Go to: `http://pytorch.org/tutorials/beginner/deep_learning_60min_blitz.html` and complete the following tutorials:

- *What is PyTorch?*
- *Autograd: automatic differentiation*
- *Neural Networks*
- *Training a classifier*

The final tutorial will leave you with a network for classifying the CIFAR-10 dataset, which is where this problem starts. Just following these tutorials could take a number of hours but they are excellent, so start early. After completing them, you should be 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`); if there is any doubt of their inputs/outputs or whether the layers include an offset or not, consult the API: `http://pytorch.org/docs/master/`.
A few preliminaries:

- Using a GPU may considerably speed up computations but it is not necessary for these small networks (one can get away with using one's laptop).

- Conceptually, each network maps an image $x^{\text{in}} \in \mathbb{R}^{32\times32\times3}$ (3 channels for RGB) to an output layer $x^{\text{out}} \in \mathbb{R}^{10}$ where the image's class label is predicted as $\arg\max_{i=0,\ldots,9} x_i^{\text{out}}$. An error occurs if the predicted label differs from its true label.

- In this problem, the network is trained via cross-entropy loss, the same loss we used for multi-class logistic regression. Specifically, for an input image and label pair (x^{in}, c) where $c \in \{0, 1, \ldots, 9\}$, if the network's output layer is $x^{\text{out}} \in \mathbb{R}^{10}$, the loss is:

$$-\log\left(\frac{\exp(x_c^{\text{out}})}{\sum_j \exp(x_j^{\text{out}})}\right).$$

- For computational efficiency reasons, this particular network considers *mini-batches* of images per training step meaning the network actually maps $B = 4$ images per feed-forward so that $\tilde{x}^{\text{in}} \in \mathbb{R}^{B\times32\times32\times3}$ and $\tilde{x}^{\text{out}} \in \mathbb{R}^{B\times10}$. This is ignored in the network descriptions below but it is something to be aware of.

- The cross-entropy loss for a neural network is, in general, non-convex. This means that the optimization method may converge to different local minima based on different hyperparameters of the optimization procedure (e.g., stepsize). Usually one can find a good solution if these hyperparameters are just observed carefully. The process of training new configurations to choose the best hyperparameters is known as *hyperparameter selection* and will be discussed more formally later (you may converge quickly or not).

- The training method used in this example uses a form of stochastic gradient descent (SGD) that uses a technique called *momentum* which incorporates scaled versions of previous gradients into the current descent direction[2]. Practically speaking, momentum is another optimization hyperparameter in addition to the step size. If this bothers you, you can obtain all the same results using regular stochastic gradient descent.

- We will not be using a validation set for this exercise. Hyperparameters like network architecture and step size should be chosen based on the performance on the test set. This is very bad practice for all the reasons we have discussed over the quarter, but we aim to make this exercise as simple as possible.

- You should modify the training code such that at the end of each epoch (one pass over the training data) you compute and print the training and test classification accuracy (you may find the running calculation that the code initially uses useful to calculate the training accuracy).

- While one would usually train a network for hundreds of epochs for it to converge, this can be prohibitively time consuming so feel free to train your networks for just a dozen or so epochs.

You will construct a number of different network architectures and compare their performance. For all, it is highly recommended that you copy and modify the existing (working) network you are left with at the end of the tutorial *Training a classifier*. For all of the following perform a hyperparameter selection (manually by hand, random search, etc.) using the test set, report the hyperparameters you found, and plot the training and test classification accuracy as a function of iteration (one plot per network). **Highly sub-optimal hyperparameter choices that lead to drastically worse error rates than your peers will result in points off (but don't over do it).**

Here are the network architectures you will construct and compare.

1. Fully connected output, 0 hidden layers (logistic regression): we begin with the simplest network possible that has no hidden layers and simply linearly maps the input layer to the output layer. That is, conceptually it could be written as

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

where $x^{\text{out}} \in \mathbb{R}^{10}$, $x^{\text{in}} \in \mathbb{R}^{32\times32\times3}$, $W \in \mathbb{R}^{10\times3072}$, $b \in \mathbb{R}^{10}$ where $3072 = 32 \cdot 32 \cdot 3$. For a tensor $x \in \mathbb{R}^{a\times b\times c}$, we let $\text{vec}(x) \in \mathbb{R}^{abc}$ be the reshaped form of the tensor into a vector (in an arbitrary but consistent pattern).

2. Fully connected output, 1 fully connected hidden layer: we will have 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 nonlinearity applied to the hidden layer will be the relu function $\text{relu}(x) = \max\{0, x\}$ (elementwise). Conceptually, one could write this network as

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

where $W_1 \in \mathbb{R}^{M\times 3072}$, $b_1 \in \mathbb{R}^M$, $W_2 \in \mathbb{R}^{10\times M}$, $b_2 \in \mathbb{R}^{10}$.

3. Fully connected output, 1 convolutional layer with max-pool: for a convolutional layer W_1 with individual filters of size $p \times p \times 3$ and output size M (reasonable choices are $M = 100$, $p = 5$) we have that $\text{Conv2d}(x^{\text{in}}, W_1) \in \mathbb{R}^{(33-p)\times(33-p)\times M}$. Each convolution will have its own offset b_1 to each of the output pixels of the convolution; we denote this as $\text{Conv2d}(x^{\text{in}}, W_1) + b_1$ where b_1 is parameterized in $\mathbb{R}^M$. We will then apply a relu (relu doesn't change the tensor shape) and pool. If we use a max-pool of size N (a reasonable choice is $N = 14$ to pool to 2×2 with $p = 5$) we have that

$$\text{MaxPool}(\text{relu}(\text{Conv2d}(x^{\text{in}}, W_1) + b_1)) \in \mathbb{R}^{3\times 3\times M}.$$

We will then apply a fully connected layer to the output to get a final network given as

$$x^{\text{out}} = W_2\text{vec}(\text{MaxPool}(\text{relu}(\text{Conv2d}(x^{\text{in}}, W_1) + b_1))) + b_2.$$

where $W_2 \in \mathbb{R}^{10\times M(33^2-p^2)}$, $b_2 \in \mathbb{R}^{10}$. The parameters M, p, N (in addition to the step size and momentum) are all hyperparameters.

4. **(Extra credit)** Returning 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 many epochs to achieve a **test accuracy of at least 87%**.

The number of hyperparameters to tune in the last exercise combined with the slow training times will hopefully give you a taste of how difficult it is to construct good performing networks. It should be emphasized that the networks we constructed are *tiny*; typical networks have dozens of layers, each with hyperparameters to tune. Additional hyperparameters you are welcome to play with if you are so interested: replacing relu $\max\{0, x\}$ with a sigmoid $1/(1+e^{-x})$, max-pool with average-pool, and experimenting with batch-normalization or dropout.