Homework 3 - Introduction to Machine Learning for Engineers

kipngeno koech - bkoech

March 14, 2025

1 Dimensionality of K-Nearest Neighbors

When the number of features d is large, the performance of k-nearest neighbors, which makes predictions using only observations that are near the test observation, tends to degrade. This phenomenon is known as the *curse of dimensionality*, and it ties into the fact that non-parametric approaches often perform poorly when d is large.

1. Suppose that we have a set of training observations, each corresponding to a one-dimensional ($d = 1$) feature, X. We assume that X is uniformly (evenly) distributed on $[0, 1]$. Associated with each training observation is a response value.Suppose that we wish to predict a test observation x's response using only training observations that are within 10% of the range of x closest to that test observation. In other words, if $x \in [0.05, 0.95]$ then we will use training observations in the range $[x - 0.05, x + 0.05]$, as shown in Figure 1 when $x = 0.6$. When $x \in [0, 0.05)$ we use the range $[0, 0.1]$, and when $x \in (0.95, 1]$ we use training observations in the range $[0.9, 1]$. Figure 1 shows this range for $x = 0.02$.On average (assuming x is uniformly distributed on $[0, 1]$), what fraction of the available observations will we use to make the prediction?

   - The fraction of the available observations will be the fraction of the range of x that is within 10% of the range of x.
   - The range of x that is within 10% of the range of x is $[x - 0.1, x + 0.1]$.
   - The fraction of the range of x that is within 10% of the range of x is $0.1 + 0.1 = 0.2$.
   - Therefore, the fraction of the available observations that we will use to make the prediction is 0.2.
   - This is equivalent to 20% of the available observations.
   - Therefore, on average, we will use 20% of the available observations to make the prediction.
   - This is because the range of x that is within 10% of the range of x is 0.2.
   - This is equivalent to 20% of the available observations.

2. Now suppose that we have a set of observations, each corresponding to two features, X_1 and X_2 (i.e., $d = 2$). We assume that (X_1, X_2) are uniformly distributed on $[0, 1] \times [0, 1]$. We wish to predict the response of a test observation (x_1, x_2) using only training observations that are within 10% of the range of x_1 and within 10% of the range of x_2 closest to that test observation. For instance, in order to predict the response for a test observation with $x_1 = 0.6$ and $x_2 = 0.04$, we will use training observations (X_1, X_2) such that $X_1 \in [0.55, 0.65]$ and $X_2 \in [0, 0.1]$.

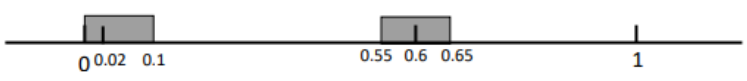

Figure 1: Range of observation, $d = 1$

Figure 1: Primal Formulation of SVM

3. On average, assuming x_1 and x_2 are each uniformly distributed on $[0, 1]$, what fraction of the available observations will we use to make the prediction?

4. Now suppose that we have a set of training observations on $d = 100$ features. Again, the observations are uniformly distributed on each feature, and again each feature ranges in value from 0 to 1. We wish to predict a test observation's response using observations within the 10% of each feature's range that is closest to that test observation. What fraction of the available observations will we use to make the prediction?

5. Using your answers to parts a–c, argue that a drawback of k-nearest neighbors when d is large is that there are very few training observations "near" any given test observation.

6. Now suppose that we wish to make a prediction for a test observation by creating a d-dimensional hypercube centered around the test observation that contains, on average, 10% of the training observations. For $d = 1$, 2, and 100, what is the length of each side of the hypercube? How does your answer change as d increases, and what does this imply for the accuracy of k-nearest neighbors when d is large?

   Note: A hypercube is a generalization of a cube to an arbitrary number of dimensions. When $d = 1$, a hypercube is simply a line segment; when $d = 2$, it is a square; and when $d = 100$, it is a 100-dimensional cube.

2 Decision Trees

You obtained the following data from interviewing 15 people on the street. Based on a person's relationship status, age, education level and income, you can now build a decision tree to predict a person's phone usage.

Relationship Status	Age	Education	Income	Phone Usage
Single	>25	University	≤50K	Low
In a relationship	≤25	College	≤50K	Medium
Single	>25	University	>50K	Low
Married	≤25	University	≤50K	High
Single	>25	University	>50K	Low
Married	>25	College	≤50K	Medium
In a relationship	≤25	College	>50K	Medium
In a relationship	>25	High School	≤50K	Low
Married	>25	University	≤50K	High
Single	>25	High School	>50K	Low
In a relationship	≤25	College	>50K	Medium
In a relationship	>25	High School	≤50K	Low
Married	>25	University	≤50K	High
Single	≤25	High School	>50K	Low
In a relationship	≤25	College	>50K	Medium

Figure 2: Primal Formulation of SVM

1. What is the entropy of Phone Usage? (Calculate entropy in log2 base and round to 4 decimal places)
2. Find the information gain (IG) from each feature (relationship status, age, education levarianceel and income). Which feature should be chosen at the root of the tree? Show your calculations for the information gain (IG) and explain your choice in a sentence. (Calculate entropy in log2 base and round to 4 decimal places)
3. Use the root you found in part (b) to determine the rest of the nodes in the decision tree for the abovariancee data. Draw the full decision tree, i.e., keep splitting the nodes until further splits do not lead to any information gain (you may not need all the features for this). For each split, show your working on why you chose this feature based on information gain.

3 Random Forests

Consider the data samples in Table 1, which record whether a person gets sick, together with features about their age (A), social distancing (S), and hand-washing frequency (H). Our goal is to learn a random forest to predict whether the person is prone to getting sick.

Sample	**Age (A)**	**Social Distancing (S)**	**Hand Washing (H)**	**Getting Sick**
1	Old	Yes	Yes	No
2	Old	Yes	No	No
3	Old	Yes	Yes	No
4	Young	No	No	Yes
5	Young	Yes	Yes	No
6	Old	No	No	Yes
7	Old	No	Yes	Yes
8	Old	No	No	Yes

Table 1:

Instead of using all features to build a single decision tree, we decide to use the random forest algorithm. Specifically, we will build three trees, each using only two features, and then combine their outcomes for the final prediction.

1. Build 3 decision trees using two features out of the three for each tree. Use information gain to decide the feature to split on. Use majority voting if all the samples in a leaf node do not have the same label.
2. Given a new data point with the features (A = old, S = yes, H = no), use the trees you learned from part (a) to predict whether this person will get sick.
3. Briefly (in 1-2 sentences) comment on the advantage of random forests over decision trees from the perspective of bias-variance trade-off.

4 Adaboost

Recall the AdaBoost algorithm for classification of a training dataset (x_n, y_n) for $n = 1, \ldots, N$ and $y \in \{-1, 1\}$. AdaBoost sequentially combines T weak classifiers $h_1(x), h_2(x), \ldots, h_T(x)$ to build one strong classifier $\text{sign}(f_T(x))$. The steps of the algorithm are as follows:

- **Initialize the weights:** $w_1(n) = \frac{1}{N}$, for all points $n = 1, 2, \ldots, N$.
- **For** $t = 1, \ldots, T$**:**
  - **a.** Learn classifier $h_t(x)$ that minimizes the error:
  $$\epsilon_t = \sum_{n=1}^{N} w_t(n) \cdot \mathbb{1}(y_n \neq h_t(x_n)).$$
  - **b.** Update the contribution:
  $$\beta_t = \frac{1}{2} \log_2 \left(\frac{1 - \epsilon_t}{\epsilon_t} \right).$$
  - **c.** Update weights of training points:
  $$w_{t+1}(n) \propto w_t(n) \cdot 2^{-\beta_t y_n h_t(x_n)},$$
  where the weights are normalized to ensure that $\sum_{n=1}^{N} w_{t+1}(n) = 1$.
- **Return the final classifier:** $\text{sign}(f_T(x))$, where
$$f_T(x) = \sum_{t=1}^{T} \beta_t h_t(x) \quad \text{for a given } x.$$

Note: The difference from the algorithm described during the lecture is that we have replaced $\log_e$ by $\log_2$ and $e^{-\beta_t}$ by $2^{-\beta_t}$ everywhere.

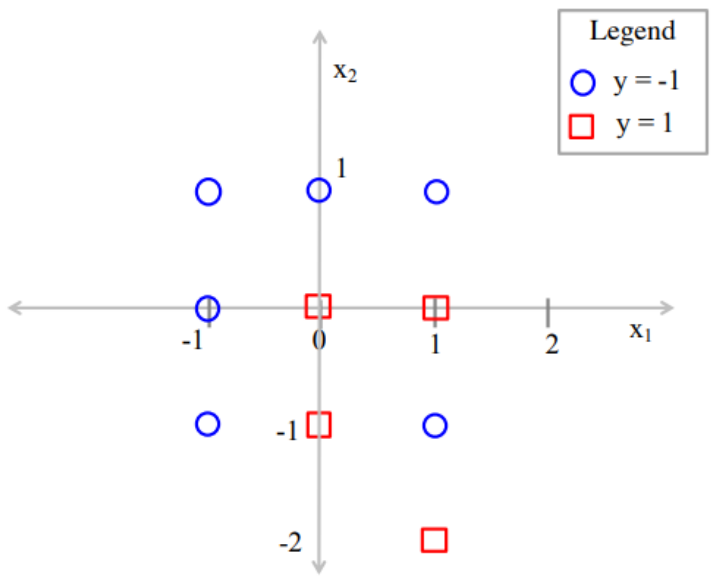

Figure 3: Primal Formulation of SVM

Consider the training dataset with $N = 10$ points and two-dimensional features $x = (x_1, x_2)$ as shown in the figure above. In this problem, we will use a binary linear decision boundary as the base classifier and perform two iterations of AdaBoost.

1. Starting with equal weights $w_1(n) = \frac{1}{10}$ for all $N = 10$ points, which of the decision boundaries shown below gives the lowest error ϵ_1? Select one of the following answers:
   (a) Predict $y = 1$ if $x_2 \leq 0.5$.
   (b) Predict $y = 1$ if $-x_1 + x_2 \leq -0.5$.
   (c) Predict $y = 1$ if $-x_1 + x_2 \leq 0.5$.
2. Compute the error ϵ_1 and the contribution β_1 of the decision boundary that you chose in part (a) above.
3. Compute the updated and normalized weights $w_2(n)$ of each of the data points as follows:
4. Suppose you are given one more weak classifier that predicts $y = 1$ if $1.5x_1 + x_2 \leq 0.5$, as shown in the figure below. Compute the contribution β_2 of this classifier for the updated set of weights $w_2(n)$. Use the approximation $\log_2 3 \approx 1.6$.
5. Combine the new classifier with the classifier that you obtained in part (a). Specify the label $y \in \{-1, 1\}$ predicted by this combined classifier for each data point. Do you observe any change in the prediction, compared to the prediction from only using part (a)'s classifier?
6. If we continue adding more classifiers, each with error less than 0.5, how does the training error of the combined classifier $f_T(x)$ change? Briefly explain your answer in 1-2 sentences.

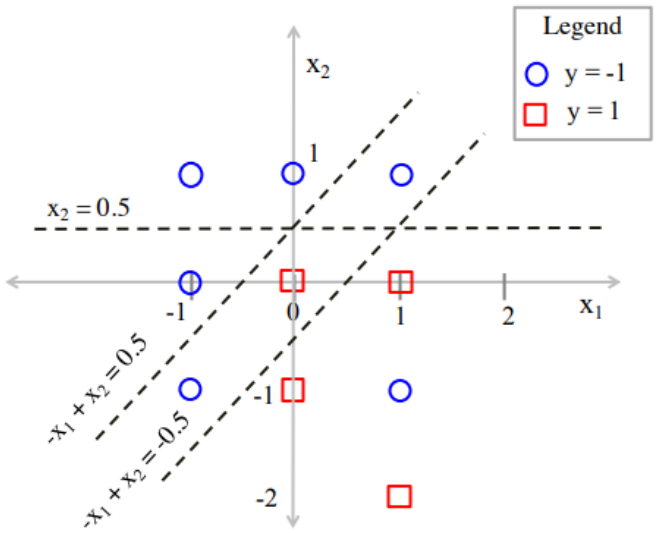

Figure 4: Primal Formulation of SVM

n	**Coordinates**	$w_2(n)$
1.	$(-1, 1)$	
2.	$(0, 1)$	
3.	$(1, 1)$	
4.	$(-1, 0)$	
5.	$(0, 0)$	
6.	$(1, 0)$	
7.	$(-1, -1)$	
8.	$(0, -1)$	
9.	$(1, -1)$	
10.	$(1, -2)$	

Table 2:

5 Neural Networks

Below is a deep network with inputs x_1, x_2. The internal nodes and activation functions are as shown in the figure and equations below. All variables are scalar values, and $\exp(x)$ refers to the function e^x. The activation functions of the nodes h_1, h_2, h_3 are ReLU (i.e., $r_1 = \max(h_1, 0)$ etc.), for node s_1 we have $s_1 = \max(r_2, r_3)$, and for the other nodes:

$$y_1 = \frac{\exp(r_1)}{\exp(r_1) + \exp(s_1)}, \quad y_2 = \frac{\exp(s_1)}{\exp(r_1) + \exp(s_1)}, \quad z = y_1 + y_2.$$

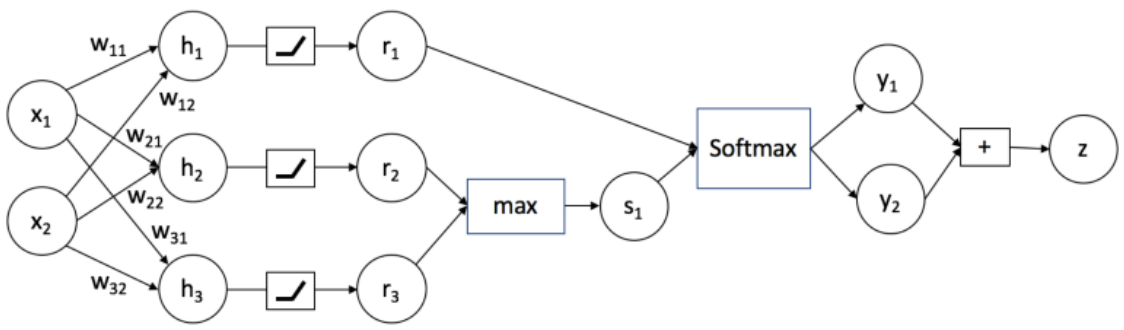

Figure 5: Primal Formulation of SVM

(a) **Forward propagation**Now, Given $x_1 = 1$, $x_2 = -2$, $w_{11} = 6$, $w_{12} = 2$, $w_{21} = 4$, $w_{22} = 7$, $w_{31} = 5$, $w_{32} = 1$, compute the values of the internal nodes (shown in the table below). You may leave e in your answer.

h_1	h_2	h_3	r_1	r_2	r_3	s_1	y_1	y_2	z

Table 3: Values of internal nodes after forward propagation.

(b) **Bounds on Variables**

- Find the range of feasible values for y_1.
- Find the range of feasible values for z.

(c) **Backpropagation** Compute the gradient expressions shown in the table below analytically. The answer should be an expression that may include any of the nodes in the network $(x_1, x_2, h_1, h_2, h_3, r_1, r_2, r_3, s_1, y_1, y_2, z)$ or weights $(w_{11}, w_{12}, w_{21}, w_{22}, w_{31}, w_{32})$.

Gradient	Expression
$\frac{\partial h_1}{\partial w_{12}}$	
$\frac{\partial h_1}{\partial x_1}$	
$\frac{\partial r_1}{\partial h_1}$	
$\frac{\partial y_1}{\partial r_1}$	
$\frac{\partial y_1}{\partial s_1}$	
$\frac{\partial z}{\partial y_1}$	
$\frac{\partial z}{\partial x_1}$	
$\frac{\partial s_1}{\partial r_2}$	

Table 4: Gradient expressions for backpropagation.

Figure 2: Sample images from the MNIST dataset.

Figure 6: Primal Formulation of SVM

6 Neural Networks for MNIST Digit Recognition

The classic MNIST dataset (Figure 2) consists of 28x28 pixel (784 features) grayscale images of handwritten digits 0-9 (10 classes). In this problem, you will implement a neural network to classify the MNIST dataset using raw pixels as features. Specifically, you will implement a type of architecture called a Multi-Layered Perceptron (MLP), which consists of several Dense layers (i.e. "Perceptrons") connected sequentially, with nonlinear activation functions after each layer. You will then train this network using the Categorical Cross-Entropy loss function.

Getting Started Download the hw3q6.zip file, which contains starter code for this question, as well as the MNIST train split (mnist.npz) and the MNIST test split with its labels removed (mnist test.npz). For this lab, you will need to install Python and Numpy. While our autograder will use Python 3.6.9, you should be able to use any recent version of Python 3 and Numpy, though you should avoid the newest Python features such as dataclasses and the walrus operator. We also recommend you install tqdm (progress bar) and Pandas (tabular data), which will be available on the autograder as well.

6.1 Gradient Derivation [8 points]

We will begin by deriving the gradients we will need to implement backpropagation. You will derive backward passes for two types of layers: the Exponential Linear Unit (ELU) activation, and a Dense (Fully Connected) layer, then use these derivations to implement the backpropagation algorithm for your neural network.

Backpropagation

Let our feed-forward (also referred to as "sequential") neural network be described by a series of functions $f_1, \ldots, f_n$ with input x_0, output x_n, loss $L = L(x_n, y^*)$ where y^* is the true label, and the feed-forward relationship:

$$x_k = f_k(w_k, x_{k-1}).$$

In our model, these functions $f_1, f_2, \ldots, f_n$ will represent our Dense layers and activation functions, and are referred to as modules in our code.

In order to perform gradient descent, we need to compute the gradient for each parameter $\frac{\partial L}{\partial w_k}$, which we can express recursively using the chain rule as:

$$\frac{\partial L}{\partial w_k} = \frac{\partial L}{\partial x_k} \cdot \frac{\partial x_k}{\partial w_k},$$
$$\frac{\partial L}{\partial x_{k-1}} = \frac{\partial L}{\partial x_k} \cdot \frac{\partial x_k}{\partial x_{k-1}}.$$

where $\frac{\partial L}{\partial x_k}$ are the gradients flowing to the previous layer, and $\frac{\partial x_k}{\partial x_{k-1}}$, $\frac{\partial x_k}{\partial w_k}$ are the Jacobians of layer f_k with respect to x_{k-1} and w_k, respectively. Note that since $\frac{\partial L}{\partial x_k}$ is the gradient of a scalar L with respect to a column vector x_k, this gradient is a row vector. And $\frac{\partial x_k}{\partial w_k}$ is the gradient of a vector with respect to a vector, which is a matrix.

Softmax Cross Entropy

This layer combines the Softmax function together with the Cross Entropy Function. The softmax function for the output layer $x_n = \begin{bmatrix} x_n^{(1)} & \dots & x_n^{(D)} \end{bmatrix}$ is described by:

$$x_n^{(i)} = \frac{\exp(x_{n-1}^{(i)})}{\sum_{j=1}^{D} \exp(x_{n-1}^{(j)})},$$

for input x_{n-1}, and the Cross Entropy loss function is given by:

$$L(x_n, y^*) = -\sum_{i=1}^{D} y^{*(i)} \log x_n^{(i)},$$

where x_n are the output weights and y^* are the true labels. Backward propagation through the Softmax Cross Entropy layer can be found by computing the first-order derivatives:

$$\frac{\partial L}{\partial x_{n-1}^{(i)}} = \frac{\partial}{\partial x_{n-1}^{(i)}} \left(-\sum_{i=1}^{D} y^{*(i)} \log \frac{\exp(x_{n-1}^{(i)})}{\sum_{j=1}^{D} \exp(x_{n-1}^{(j)})} \right)$$

$$= \frac{\partial}{\partial x_{n-1}^{(i)}} \left(-\sum_{i=1}^{D} y^{*(i)} x_{n-1}^{(i)} + \sum_{i=1}^{D} y^{*(i)} \log \sum_{j=1}^{D} \exp(x_{n-1}^{(j)}) \right)$$

$$= \frac{\partial}{\partial x_{n-1}^{(i)}} \left(-\sum_{i=1}^{D} y^{*(i)} x_{n-1}^{(i)} + \log \sum_{j=1}^{D} \exp(x_{n-1}^{(j)}) \right) \quad \text{(only one } y^{*(i)} \text{ is 1)}$$

$$= -y^{*(i)} + \frac{\exp(x_{n-1}^{(i)})}{\sum_{j=1}^{D} \exp(x_{n-1}^{(j)})}$$

$$= -y^{*(i)} + y_i.$$

The implementation of this layer is provided to you in `npnn/layers.py`.

Exponential Linear Unit (ELU)

The Exponential Linear Unit is a variant of the Rectified Linear Unit (ReLU) activation function that eliminates the zero gradient problem when the input is less than 0. Using ELU as an activation function tends to produce faster convergence with more accurate results than using ReLU. Unlike other activation functions, ELU has an extra α constant, which should be a positive number (a typical choice of α is 1 or 0.9).

The expression of an ELU unit is as follows:

$$\text{ELU}(x) = \begin{cases} x & \text{if } x > 0, \\ \alpha(e^x - 1) & \text{otherwise.} \end{cases}$$

Dense Layer

A Dense layer is fully connected with the input features. For a weight matrix W and bias b, the output is:

$$f(x) = Wx + b.$$

1. Find the Jacobian $\frac{\partial x_k}{\partial x_{k-1}}$ for the exponential linear unit, $x_k = \text{ELU}(x_{k-1})$.
2. Find the Jacobian $\frac{\partial x_k}{\partial x_{k-1}}$ for the Dense layer, $x_k = Wx_{k-1} + b$.
3. Find the Jacobian $\frac{\partial x_k}{\partial b}$ for the Dense layer, $x_k = Wx_{k-1} + b$.
4. Since W is two-dimensional, the Jacobian $\frac{\partial x_k}{\partial W}$ can only be expressed by matrix-vector multiplications if we flatten it to a vector. Instead, to preserve its dimension, find the gradient $\frac{\partial x_k[a]}{\partial W[b,c]}$ for indices a, b, c (so that we can write $\frac{\partial x_k}{\partial W}$ as a 3-dimensional tensor). Here, a is an index of x_k, b indexes a row of W, and c indexes a column of W.

6.2 Module Implementation [11 points]

Implement the ELU and Dense modules based on your answers in the previous part. The correctness of your implementation for parts 6.2 and 6.3 along with your prediction accuracy for part 6.4a will be checked in unit tests, where we will compare your implementation against our implementation.

Note: Do not use any Python libraries other than the ones provided in `requirements.txt`; otherwise, the autograder will not be able to run your code. In particular, you may not utilize any libraries performing automatic differentiation such as PyTorch, TensorFlow, and JAX in your submitted code, though you may use these libraries to test your implementation if you wish.

1. Complete the initialization `__init__` of the dense layer in `npnn/modules.py`. You should initialize the bias b to all zeros; W should be initialized using the Glorot Uniform initialization. In the Glorot Uniform initialization, each element is drawn from a uniform distribution $\text{Unif}(-u, u)$, where $u = \sqrt{\frac{6}{\text{dim\_in}+\text{dim\_out}}}$.
2. In `npnn/modules.py`, implement forward passes for the ELU and Dense modules.
3. In `npnn/modules.py`, implement backward passes for the ELU and Dense modules based on your answer in Problem 6.1.

6.3 Model Training Loop [8 points]

1. In `npnn/optimizer.py`, implement the `SGD.apply_gradients` method.
2. In `npnn/model.py`, implement forward and backward passes for the model in the `Sequential.forward` and `Sequential.backward` methods.
3. In `npnn/model.py`, implement the training loop in `Sequential.train`, and the testing method in `Sequential.test`. To help you organize your implementation, we have provided the signature of categorical cross-entropy and categorical accuracy functions for you to use.

6.4 Training and Evaluation [8 points]

In order to train your neural network, we have loaded the MNIST dataset included with the starter code (`mnist.npz`). We have randomly split the dataset into a training dataset with 50,000 elements and a validation dataset with 10,000 elements. Note, we are splitting the MNIST dataset into train and validation and leaving the test set explicitly for testing the model's predictions after completing training. Next, create a neural network with 3 Dense layers with 256, 64, and 10 units, respectively. The first two Dense layers should be followed by an ELU activation with $\alpha = 0.9$, and the last Dense layer should be followed by the `SoftmaxCrossEntropy` module. See Figure 3 for an illustration of the neural network structure.

When correctly vectorized, your implementation should take fewer than 10 seconds to perform a single epoch during training; see the module docstring for more expected runtime details. While you will not be graded on runtime, excessive runtime may indicate the presence of errors in your code, and it may impact your ability to debug your code and complete experiments in a timely manner.

Note: At this point, you should have implemented all functions or methods annotated with `# TODO` or `raise NotImplementedError()` except for the extra credit implementation of the Adam optimizer.

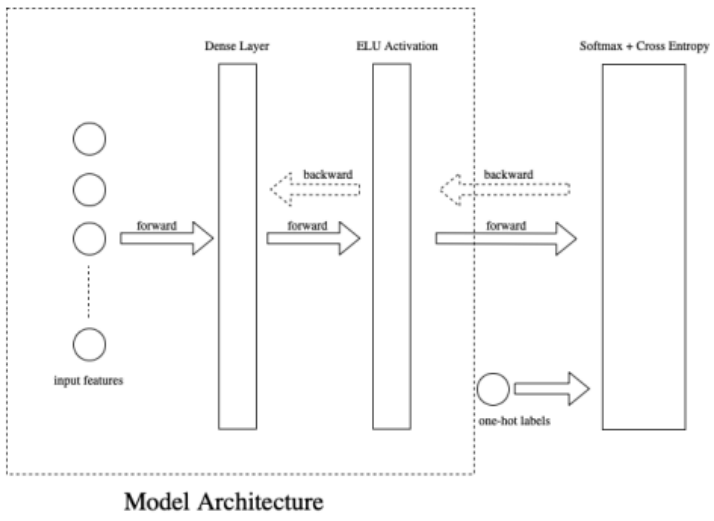

Figure 3: Neural Network Training Architecture.

Figure 7: Primal Formulation of SVM

6.4 Training and Evaluation [8 points]

1. **[2 points]** Train this network for 20 epochs with a learning rate of 0.1 and a batch size of 32, evaluating the network on your validation split after each epoch. Then, plot the train and validation accuracy after each epoch during training.

   You should submit two plots along with your answer: one showing a training loss curve and a validation loss curve with respect to the training epochs, and one showing a training accuracy curve and a validation accuracy curve with respect to the training epochs. Make sure to label both curves!

2. **[3 points]** Test your trained network on the test dataset (`mnist_test.npz`), and submit your predictions to Gradescope. You will receive credit if your test accuracy is greater than 97%. **Note:** The `mnist_test.npz` dataset does not include the true labels. You can use your validation accuracy to estimate your test accuracy in advance (as long as you perform your train-val split in an unbiased manner).

3. **[3 points]** Repeat the training procedure using learning rates of $[0.05, 0.1, 0.2, 0.5, 1.0]$, and plot the lowest validation loss obtained for each learning rate. What is the effect of changing the learning rate? Your plot should be a line graph with the learning rate on the x-axis and the best validation loss on the y-axis. Since there is a large difference in magnitude for the learning rate, you should use the log-learning rate in your graph, and set the labels manually:

```
fig, ax = plt.subplots()
# ... draw plot here ...
ax.set_xticks([0.05, 0.1, 0.2, 0.5, 1.0])
ax.set_xticklabels([0.05, 0.1, 0.2, 0.5, 1.0])
```

6.5 Bonus: Adam [+5 points]

The Adam (Adaptive Moment) optimizer is a popular SGD variant (third most popular after SGD and Momentum by citations, according to this paper) that adds "momentum" to the gradients and tries to normalize the gradients by their current magnitude. Adam defines two values for each parameter, m and v, which are initialized by zero and updated by the relationship:

$$m_{t+1} = \beta_1 m_t + (1 - \beta_1) g_t,$$

$$v_{t+1} = \beta_2 v_t + (1 - \beta_2) g_t^2,$$

where g_t is the gradient at time step t. Then, the final parameter update is defined by:

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}} + \epsilon} \hat{m},$$

where $\hat{m} = \frac{m_t}{1-\beta_1}$ and $\hat{v} = \frac{v_t}{1-\beta_2}$. These updates are controlled by four hyperparameters: β_1, the momentum decay constant; β_2, the second moment decay constant; η, the learning rate; and ϵ, a small constant added to the denominator.

Implement the Adam optimizer in `npnn/optimizer.py`, and tune the learning rate, setting the other hyperparameters to $\beta_1 = 0.9$, $\beta_2 = 0.999$, $\epsilon = 10^{-7}$. How does a properly-tuned Adam optimizer compare to SGD? How does the sensitivity of Adam to its learning rate compare to SGD?