ENM360 Midterm exam (Fall 2020)

Instructor: Paris Perdikaris

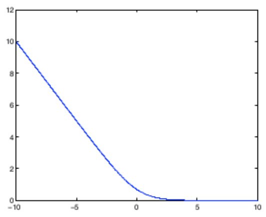
Thursday, October 27, 2020

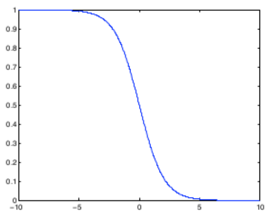
1. **[5 pts]** During the midterm your classmate whispers the answer to a problem, call it n, to two students George (G) and Yibo (Y) independently. Due to your classmate speaking in a very low voice, each student imperfectly and independently draws a conclusion about what the answer result number was. G thinks that heard the number n_G and and Y thinks that he heard the number n_Y. Are n_G and n_Y marginally independent but conditionally independent given the true number n? Provide an explanation to your answer.

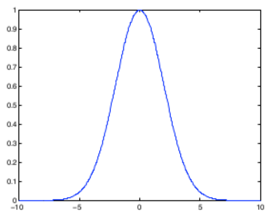
   Solution: Yes they are. Knowledge of n_G value tells us something about n_Y, therefore $P(n_G|n_Y) \neq P(n_G)$ hence they are marginally dependent, but given n, n_G and n_Y are determined independently.

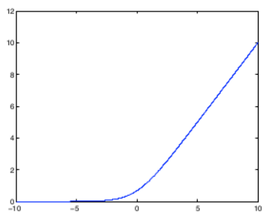
2. **[15 pts]** Let's say that you are considering a model to solve a problem that requires classifying the data into different classes. So, you want to find some weights (parameters) w the are needed in order to minimize the considered loss function $\sum_i \mathcal{L}(\phi^i(y(x^i))$ over the training set. For example consider the linear classifier $y(x) = w_0 + \sum_{j=1}^d w_j x_j$ and $\phi(y(x)) = \phi(w_0 + \sum_{j=1}^d w_j x_j)$.

   (a) **[5 pts]**You are given some potential loss functions. Which one (or more than one) are appropriate to use in this task? For the ones that are not appropriate, provide a short explanation why this is the case. In general, what conditions have to be satisfied for $\mathcal{L}$ to be an appropriate loss function? In the figures below, the x axis represents $\phi(y(x))$ and the y axis $\mathcal{L}((F(x))$.


(a)


(b)


(c)


(d)

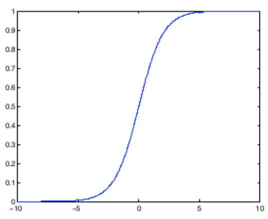

(e)

Figure 1: Potential Loss functions.

Solution: (a) and (b) are appropriate to use in classification. In (c), there is very little penalty for extremely misclassified examples, which correspond to very negative $\phi(y(x))$. In (d) and (e), correctly classified examples are penalized, whereas misclassified examples are not. In general, $\mathcal{L}$ should approximate the 0-1 loss, and it should be a non-increasing function of $\phi(y(x))$.

(b) **[5 pts]** Out of the loss functions given above which one (only one) do you think that is more robust to outliers (meaning a data point that differs significantly from other observations)? Provide a simple justification to your answer (*For outliers, $\phi(y(x))$ often takes a high negative value*).

Solution: (b) is more robust to outliers. For outliers, $\phi(y(x))$ is often very negative. In (a), outliers are heavily penalized. So the resulting classifier is largely affected by the outliers. On the other hand, in (b), the loss of outliers is bounded. So the resulting classifier is less affected by the outliers, and thus more robust.

(c) **[5 pts]** Suppose that your function is given by the function $\mathcal{L}(\phi(y(x))) = \frac{1}{1+exp(\phi(y(x)))}$, where $y(x) = w_0 + \sum_{j=1}^{d} w_j x_j$. You choose a gradient descent optimizer to obtain the optimal parameters w_0 and w_j. Derive the update rules that you will use to find these parameters.

Solution: To obtain the parameters for $y(x)$ we need to optimize

$$\sum_i \mathcal{L}(\phi^i(y(x^i))) = \frac{1}{1+exp(\phi^i(y(x^i)))} = \frac{1}{1+exp(w_0+\sum_{j=1}^{d} w_j x_j)}$$

. So, we are doing the following:

$$\frac{\partial}{\partial w_0}\sum_i \mathcal{L}(\phi^i(y(x^i))) = \frac{\partial}{\partial w_0}\left(\frac{1}{1+exp(\phi^i(y(x^i)))}\right) = -\sum_i \frac{\phi^i exp(\phi^i(y(x^i)))}{(1+exp(\phi^i(y(x^i))))^2} \qquad k = 1, ..., d$$

$$\frac{\partial}{\partial w_k}\sum_i \mathcal{L}(\phi^i(y(x^i))) = \frac{\partial}{\partial w_k}\left(\frac{1}{1+exp(\phi^i(y(x^i)))}\right) = -\sum_i \frac{\phi^i x_k^i exp(\phi^i(y(x^i)))}{(1+exp(\phi^i(y(x^i))))^2} \qquad k = 1, ..., d$$

So the update rules are equal to:

$$w_0^{(t+1)} = w_0^t - \eta\frac{\partial}{\partial w_0}\sum_i \mathcal{L}(\phi^i(y(x^i))) = w_0^t - \eta\sum_i \frac{\phi^i exp(\phi^i(y(x^i)))}{(1+exp(\phi^i(y(x^i))))^2}$$

and

$$w_k^{(t+1)} = w_k^t - \eta\frac{\partial}{\partial w_k}\sum_i \mathcal{L}(\phi^i(y(x^i))) = w_k^t - \eta\sum_i \frac{\phi^i x_k^i exp(\phi^i(y(x^i)))}{(1+exp(\phi^i(y(x^i))))^2}$$

For every $k = 1, ..., d$

3. **[20 pts]** Consider the following Neural Network with linear activation functions. So, the output of each neuron is a constant C multiplied by the weighted sum of the inputs, i.e.

$$h^{i+1} = C(Wh^i + b)$$

(a) **[5 pts]** Can the above network be represented by a Neural Network with a single layer and neuron? If you think that this is the case, draw the equivalent network, detailing the weights and the activation functions. Otherwise explain why not.

Solution: Yes, it is possible to do this. You have to use C^2 as an activation function. And weights for X_1, $W_1 = w_1 * w_5 + w_2 * w_6$ and for X_2, $W_2 = w_3 * w_5 + w_4 * w_6$. Any answer that provides an equivalent function by a correct linear combination of the weights is correct. Every answer were the above is explained properly is also correct.

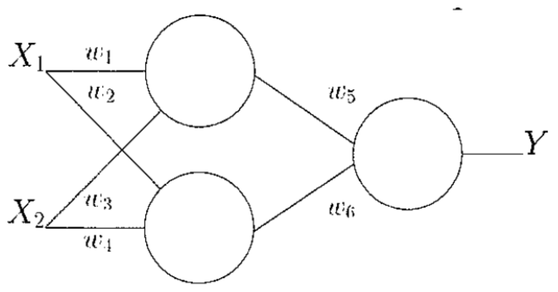


Figure 2: Neural Network.

(b) **[5 pts]** Can the a solution represented by the above network be also represented by linear regression? Provide a short explanation to your answer.

Solution: Yes, it can be done. Any function in the network above has the form:

$$Y = C^2(w_1 * w_5 + w_2 * w_6)X_1 + C^2(w_3 * w_5 + w_4 * w_6)X_2$$

or

$$Y = \beta_1 X_1 + \beta_2 X_2$$

which is linear regression on X_1, X_2 with coefficients β_1, β_2

(c) **[5 pts]** Let's say you have 2 activation functions:

- **S**: signed sigmoid activation function $S(a) = \text{sign}(\sigma(a) - 0.5) = \text{sign}(\frac{1}{1+exp(-a)} - 0.5$
- **L**: linear function $L(a) = ca$

in both cases $a = \sum_i w_i X_i$.

Assign proper activation functions to the above network (**S** or **L**) for each unit in the above graph to simulate the binary logistic regression classifier:

$$Y = argmax_y P(Y = y|X)$$

where

$$P(Y = 1|X) = \frac{exp(\beta_1 X_1 + \beta_2 X_2)}{1 + exp(\beta_1 X_1 + \beta_2 X_2)}$$

and

$$P(Y = -1|X) = \frac{1}{1 + exp(\beta_1 X_1 + \beta_2 X_2)}$$

Provide a short justification to your answer.

Solution: The neurons of the first layer would have **L** and the neuron of the last layer will have **S**

(d) **[5 pts]** Assign proper activation functions to the above network (**S** or **L**) for each unit in the above graph to simulate a classifier that combines two logistic regression classifiers, $f_1 : X \mapsto Y_1$ and $f_2 : X \mapsto Y_2$ to produce the final prediction:

$$Y = \text{sign}[a_1 Y_1 + a_2 Y_2],$$

where $f_1 = P(Y = 1|X)$ and $f_2 = P(Y = -1|X)$ from the previous question.
Provide a short justification to your answer.

Solution: The neurons of the first layer would have **S** and the neuron of the last layer will have **S**

4. **[10 pts]** Let's assume that you have 2 datasets $\mathcal{D}_1$ and $\mathcal{D}_2$. For $\mathcal{D}_1$ we have 5,000 observations and for $\mathbf{D}_2$ we have 100,000 observations. We construct the train and test set by splitting each data set with ratio 90 : 10.

   Draw two, curves by hand, for each dataset that will represent the training and the test error. The y axis of the curve should be the error and the x axis is the model complexity, meaning the number of parameters considered in the model. For the example, in neural network the number of parameters should be the number of layers and the considered number of neurons at each layer. So, you need four curves, two for each dataset. Draw all four curves at the same figure and mark clearly each curve.

   Solution: Two important things:

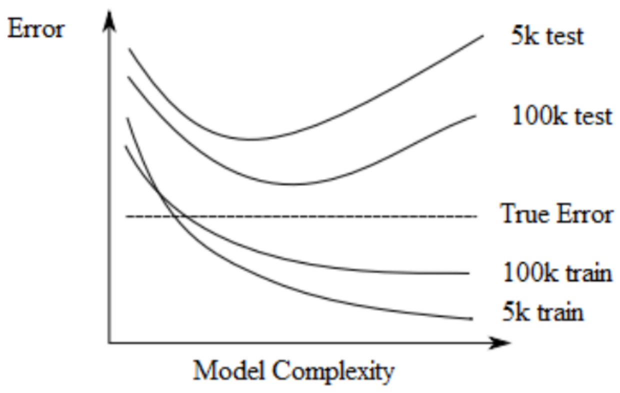


Figure 3: Curves for 2 datasets.

   a) The training error decreases as you increase the model complexity, while the test error decreases at first but then it increases, due to overfitting. So the dataset with the less data should have the best training performance (smallest training error) and the worst generalization (largest test error).
   b) Given the same model complexity, the model has larger training samples is less likely to overfit than the one with less samples. So the two curves representing 100k samples are nearer the dash line the other two curve (exactly the opposite from the above)

5. **[5 pts]** Let's assume that we drop a drunk squirrel at location s of an 1-dimensional branch of a tree, where s is drawn from a Gaussian $s \sim \mathcal{N}(\mu_s = 0, \sigma_s^2 = 2^2)$.Now the squirrel makes a step on the right by distance d, where $d \sim \mathcal{N}(0,1)$ (If d is negative the squirrel moves on the left). If we write f the final location of the squirrel, we see that $f \sim \mathcal{N}(s,1)$. Assume that d is independent of s.

   The squirrel ends up at location $f = 2$. What is the most likely location s that the squirrel landed on the branch initially (*Hint: Use properties of the multi-variate normal distribution*)?

   Solution: Tricky squirrels..

   Based on the definition you can write the probability of the start location s and final location f as:

$$\begin{pmatrix} s \\ f \end{pmatrix} \sim \mathcal{N}\left(\mu = \begin{pmatrix} 0 \\ 0 \end{pmatrix}, \Sigma = \begin{pmatrix} 4 & 4 \\ 4 & 5 \end{pmatrix}\right) \tag{1}$$

   So,

$$\mu_{s|f} = \mu_s + \frac{\Sigma_{12}}{\Sigma_{22}}(f - \mu_f) = 0 + \frac{4}{5}(f - 0) = \frac{4}{5}f$$

   Most likely s is given $f = 2$: S = $\text{argmax}_s P(s|f = 2) = \mu_{s|f=2} = \frac{4}{5} * 2 = 1.6$

6. **[10 pts]** Consider the plot below that shows the error of the algorithm on the training set and the error on the validation set for the Backpropagation algorithm training a neural network for a particular medical diagnosis problem. In the figure below the x axis represent the number of weight updates and the y axis the error. The training error decreases monotonically when the gradient descent steps are

increased, but the validation error does not. Let's say that we retrain the same neural network using exactly the same algorithm, but ten times as much data.

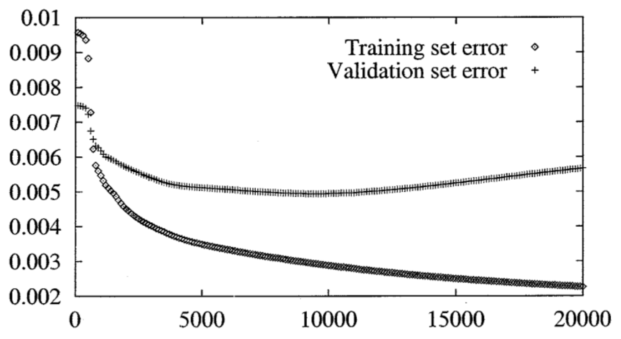


Figure 4: Error versus weight updates.

(a) **[5 pts]** Would you expect the training curve to be different? If yes, draw your expectation. Despite your answer being yes or no, provide a brief explanation of your answer.

Solution: If we are performing batch learning and keep the same learning rate, the steps will be larger and so we will learn faster (possibly unstable). This case is also problem dependent and interpretation depended, therefore whichever answer had merit and was well explained was considered correct.

(b) **[5 pts]** Would you expect the validation set curve to be different? If yes, draw your expectation. Despite your answer being yes or no, provide a brief explanation of your answer.

Solution: If we ignore the above effect (e.g. by decreasing the learning rate by a factor of 10) then the training curve would remain pretty much the same as before initially but it would end up not overfitting and so might not go down so far on the right. If we are not overfitting so much the validation curve will not increase so much. Any answer that reflects this is correct. So, again both answers have merit so if explained properly you can get full credit.

7. **[5 pts]** Briefly describe the different techniques you know for regularizing deep neural networks (*Discuss the techniques explained in class.*).

Solution:
1. Batch Normalization
2. Dropouts
3. L1 loss
4. L2 loss
5. Data augmentation.
6. Early stopping.

8. **[20 pts]** Consider the following Bayesian regression model with a Gaussian likelihood over iid input/output observations $\mathcal{D} = (\boldsymbol{X}, y)$:

$$p(y|\boldsymbol{X}, W, \gamma) = \prod_{i=1}^{N} \mathcal{N}(y_i | f(\boldsymbol{x}_i; W), \gamma^{-1}),$$

and priors over the model weights W, and the precisions λ and γ:

$$p(W|\lambda) = \mathcal{N}(w|0, \lambda^{-1}),$$
$$p(\lambda) = \text{Gam}(\lambda|\alpha_0^\lambda, \beta_0^\lambda),$$
$$p(\gamma) = \text{Gam}(\gamma|\alpha_0^\gamma, \beta_0^\gamma),$$

where $(\alpha_0^\lambda, \beta_0^\lambda, \alpha_0^\gamma, \beta_0^\gamma)$ are assumed to be known parameters, and Gam denotes the Gamma distribution.

(a) **[5 pts]** Use Bayes rule to express the posterior over the model parameters $p(W, \lambda, \gamma|\boldsymbol{X}, y)$ as a function of the likelihood and the priors.

Solution: $p(W, \lambda, \gamma|\boldsymbol{X}, y) = \frac{p(y|\boldsymbol{X}, W, \lambda, \gamma)}{p(y|\boldsymbol{X})}$ you can also expand the numerator. Both taking into acount the regularizer and not taking into acount is considered correct based on your explanation.

(b) **[5 pts]** Derive the optimization objective for training the model parameters (W, λ, γ) using maximum likelihood estimation (MLE). In other words, derive the expression for $-\log p(y|\boldsymbol{X}, W, \gamma)$ using the definition of the Gaussian probability density function:

$$p(y_i|x_i, W, \gamma) = \sqrt{\frac{\gamma}{2\pi}} \exp[-\frac{\gamma}{2}(y_i - f(x_i; W))^2]$$

Solution: $-\log p(y|\boldsymbol{X}, W, \gamma) = \frac{N}{2} \log \frac{2\pi}{\gamma} + \frac{\gamma}{2} \sum (y_i - f(x_i; W))^2$.

(c) **[10 pts]** Derive the optimization objective for training the model parameters (W, λ, γ) using maximum a-posteriori estimation (MAP). To do so, recall that the posterior over the model parameters is proportional to $p(W, \lambda, \gamma|\boldsymbol{X}, y) \propto p(y|\boldsymbol{X}, W, \gamma)p(W|\lambda)p(\lambda)p(\gamma)$, and use the following expression for computing the log of the Gamma distribution:

$$\log \mathrm{Gam}(c|\alpha, \beta) = (\alpha - 1)\log c - \beta c + \log c.$$

Solution: $L = \frac{N}{2} \log \frac{2\pi}{\gamma} + \sum (y_i - f(x_i; W))^2 + \frac{\gamma}{2} W^T W - [(\alpha_0^\lambda - 1)\log \lambda - \beta_0^\lambda \lambda + \log \lambda] - [(\alpha_0^\gamma - 1)\log \gamma - \beta_0^\gamma \gamma + \log \gamma]$.

9. **[10 pts]** Find the bugs in the following code:

Solution: be line 4: std(0) instead of 1

line 5: std(0) instead of 1

line 12: X.shape[1] not 0

line 27: -4.0 not 4]

line 31: params["weights"] not params["noise logvar"]

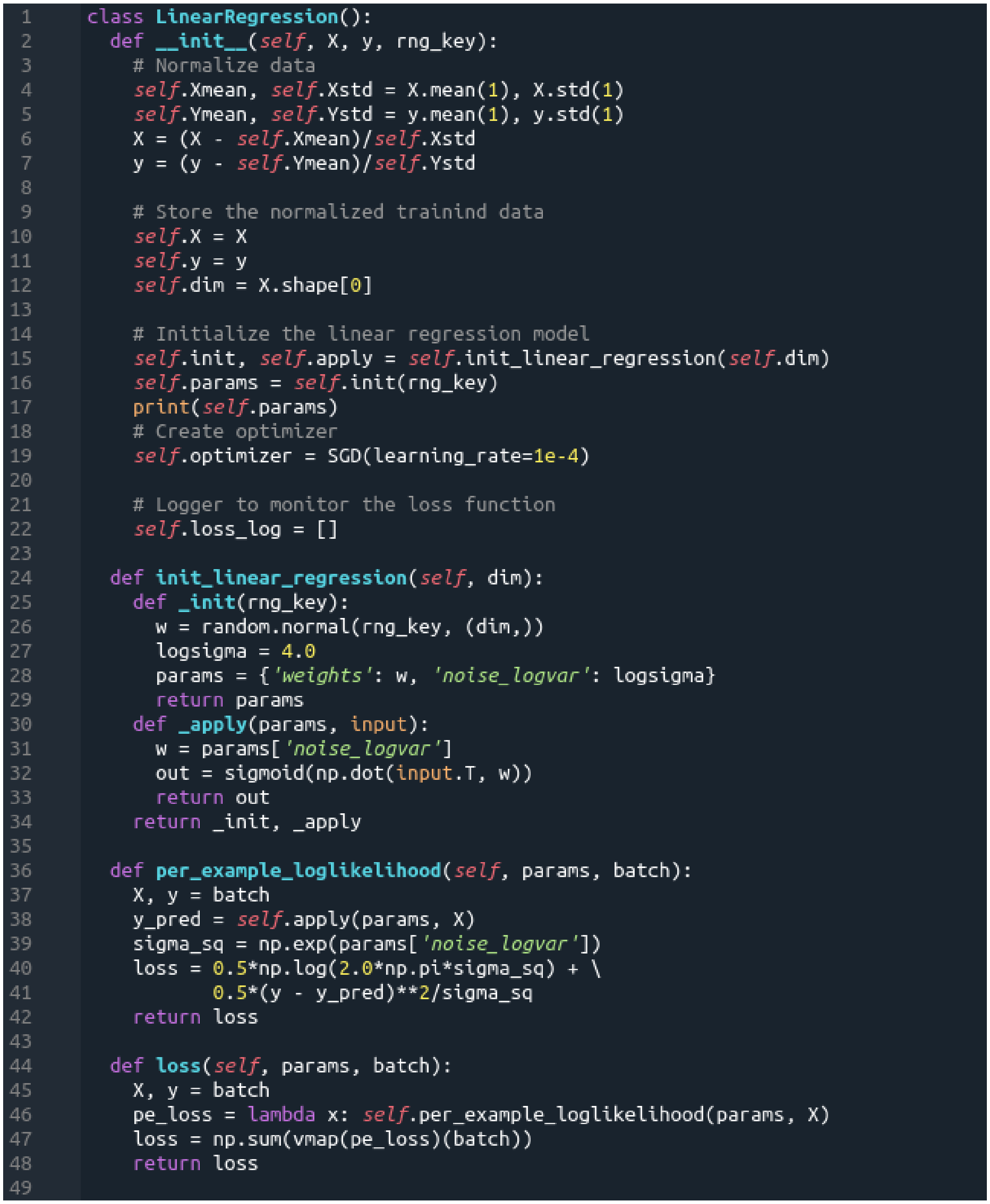
line 32: (input,w) not (input.T,w)

line 46: (params, x) not (params, X)

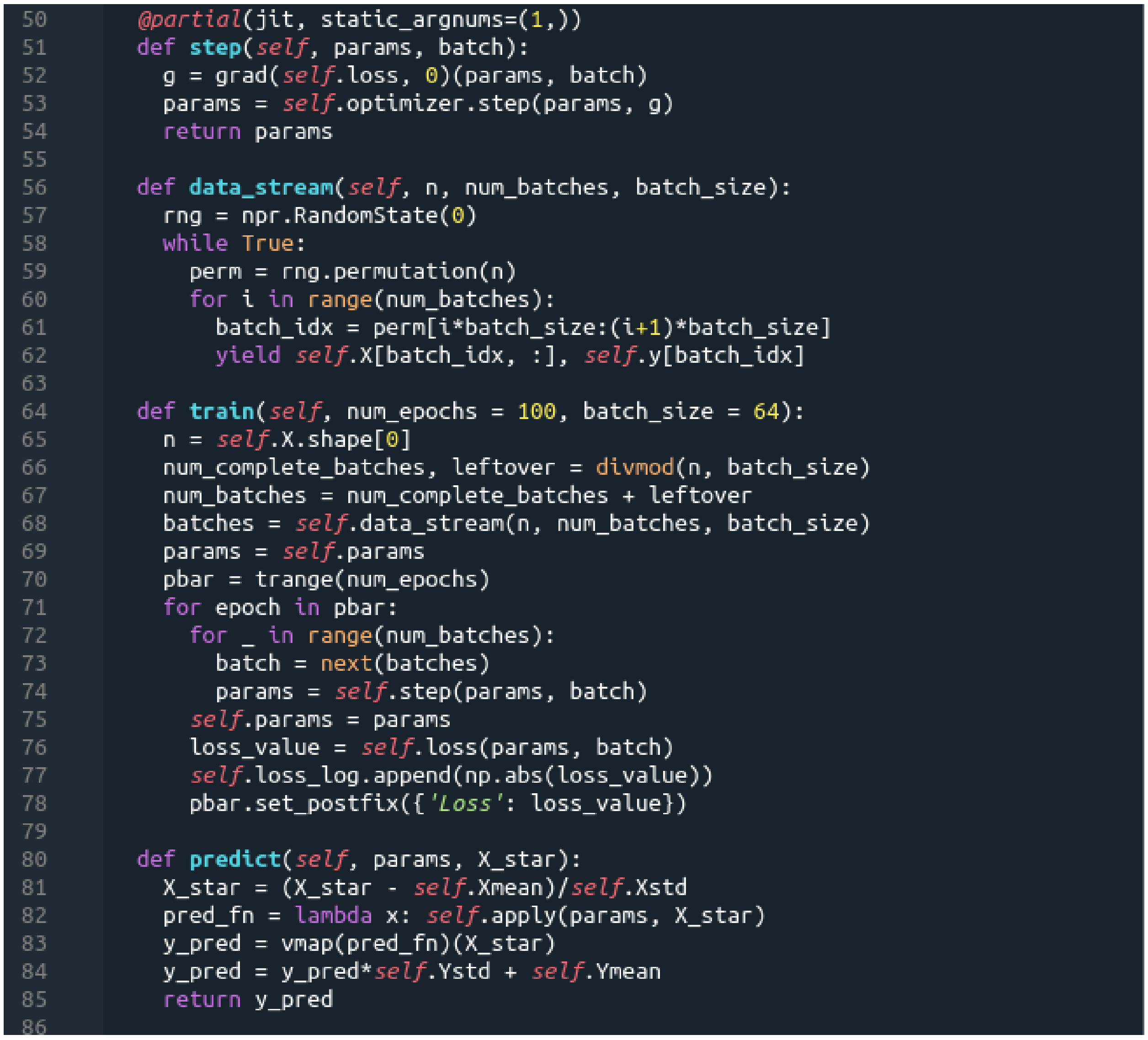
line 75: @partial(jit, static argnums=(0,)) not @partial(jit, static argnums=(1,))

line 92: boolean (leftover) not leftover

line 107: (params, x) not (params, X$_s tar$)



```python
class LinearRegression():
  def __init__(self, X, y, rng_key):
    # Normalize data
    self.Xmean, self.Xstd = X.mean(1), X.std(1)
    self.Ymean, self.Ystd = y.mean(1), y.std(1)
    X = (X - self.Xmean)/self.Xstd
    y = (y - self.Ymean)/self.Ystd

    # Store the normalized traind data
    self.X = X
    self.y = y
    self.dim = X.shape[0]

    # Initialize the linear regression model
    self.init, self.apply = self.init_linear_regression(self.dim)
    self.params = self.init(rng_key)
    print(self.params)
    # Create optimizer
    self.optimizer = SGD(learning_rate=1e-4)

    # Logger to monitor the loss function
    self.loss_log = []

  def init_linear_regression(self, dim):
    def _init(rng_key):
      w = random.normal(rng_key, (dim,))
      logsigma = 4.0
      params = {'weights': w, 'noise_logvar': logsigma}
      return params
    def _apply(params, input):
      w = params['noise_logvar']
      out = sigmoid(np.dot(input.T, w))
      return out
    return _init, _apply

  def per_example_loglikelihood(self, params, batch):
    X, y = batch
    y_pred = self.apply(params, X)
    sigma_sq = np.exp(params['noise_logvar'])
    loss = 0.5*np.log(2.0*np.pi*sigma_sq) + \
           0.5*(y - y_pred)**2/sigma_sq
    return loss

  def loss(self, params, batch):
    X, y = batch
    pe_loss = lambda x: self.per_example_loglikelihood(params, X)
    loss = np.sum(vmap(pe_loss)(batch))
    return loss

```



```
  @partial(jit, static_argnums=(1,))
  def step(self, params, batch):
    g = grad(self.loss, 0)(params, batch)
    params = self.optimizer.step(params, g)
    return params

  def data_stream(self, n, num_batches, batch_size):
    rng = npr.RandomState(0)
    while True:
      perm = rng.permutation(n)
      for i in range(num_batches):
        batch_idx = perm[i*batch_size:(i+1)*batch_size]
        yield self.X[batch_idx, :], self.y[batch_idx]

  def train(self, num_epochs = 100, batch_size = 64):
    n = self.X.shape[0]
    num_complete_batches, leftover = divmod(n, batch_size)
    num_batches = num_complete_batches + leftover
    batches = self.data_stream(n, num_batches, batch_size)
    params = self.params
    pbar = trange(num_epochs)
    for epoch in pbar:
      for _ in range(num_batches):
        batch = next(batches)
        params = self.step(params, batch)
      self.params = params
      loss_value = self.loss(params, batch)
      self.loss_log.append(np.abs(loss_value))
      pbar.set_postfix({'Loss': loss_value})

  def predict(self, params, X_star):
    X_star = (X_star - self.Xmean)/self.Xstd
    pred_fn = lambda x: self.apply(params, X_star)
    y_pred = vmap(pred_fn)(X_star)
    y_pred = y_pred*self.Ystd + self.Ymean
    return y_pred
```