Midterm Group Review

CPSC 585; Spring 2021; Professor Avery

Group Members

- Mike Peralta <mikeperalta@csu.fullerton.edu>
- Thomas Ngo <tngo0508@csu.fullerton.edu>
- Adam Shirley <bsa919adam@gmail.com>
- Brian Alexander <balexander28@csu.fullerton.edu>

About This Document

We copied the somewhat “off” numbering scheme present in the “Group Exam Review” assignment, which is different from what we observed while taking the Midterm.

Question 1

Part 1 - Write the output O of the percep in closed form as a function of X and W

a. o = sign(**w** • **x**) note: **w** and **x** are vectors

Part 2 - Write the output *o* of the perceptron in closed form as a function of the individual values wi, xi, and b without using summation or vector dot product notation.

$$y = \text{sign}\ \{W_0 * b + W_1 X_1 + W_2 X_2\}$$

Part 3 - Compute the prediction *y^* for this example.

Mike guess: y = sign(0.1*1 + 0.2*2 + -0.3*3) == sign(0.1 + 0.4 + -0.9) == sign(-0.4) == -1

Part 4 - Compute the squared loss for this training example.

oss = $(y - \hat{y})^2 = (2)^2 = 4$

Part 5 - Use the perceptron update rule

$W \leftarrow W + \boldsymbol{\alpha} (y - \hat{y}) X$

$W = [0.1, 0.2, -0.3] + \boldsymbol{\alpha}*(2)*[1, 2,3] = (0.1+2\boldsymbol{\alpha}, 0.2+4\boldsymbol{\alpha}, -0.3+6\boldsymbol{\alpha})$

Question 2 - Widrow-Hoff Neuron

Part 3 - Closed form

$y = (W_0*b + W_1X_1 + W_2X_2)$

We use no activation because the identity is just the weighted sum.

Part 4 - Compute Partial dL/dX1

$\partial o/\partial x_1 = w_1$
$\partial L/\partial x_1 = \partial L/\partial o * \partial o/\partial x_1 = (5) (0.2) = 1$

Part 5 - Compute Rest of values in the gradient

$\partial o/\partial x_2 = w_2$
$\partial L/\partial x_2 = \partial L/\partial o *\partial o/\partial x_2 = (5) (-0.3) = -1.5$
$\partial L/\partial X_0 = \partial L/\partial O * \partial O/\partial X_0 = 5(0.1) = 0.5$

We're still writing the bias input as X_0 with corresponding weight as W_0

Part 6 - Compute Gradient Descent with alpha=0.1 to update weights

weights = weights -$\boldsymbol{\alpha}$ * gradient

Gradient = (0.5, 1, -1,5)
***weights* = [0.1, 0.2, -0.3] - 0.1*[0.5, 1, -1.5]**
***weights* = [0.1, 0.2, -0.3] - [0.05, 0.1, -0.15]**
***weights* = [0.05, 0.1, -0.15]**

Question 3 - Multinomial Regression Classifier

Part 4 - How many inputs?

4 inputs because 4 features

Part 5 - How many outputs?

3 outputs because we're trying to classify against 3 classes.

Part 6 - How many weights are in the network?

There should be one set of weights per class, with each set having one weight per input, plus an additional input for the bias.

So (4+1) * 3 = 15 total weights in the entire network.

Part 7 - What activation and loss functions should the network use?

Use softmax activation and cross-entropy for loss function.

Part 8 - Describe how the network predicts a class for a given feature vector x-vec

Given vector x, the multinomial logistic regression gives us the outputs that contain likelihood or probabilities of each class. Based on the probability, the prediction is the highest probability or most likely class to occur.

Question 4 - Feedforward; Too Small Hyper Parameters

Part 5 - Too Small Learning Rate (alpha)

1. learning/convergence is slow
2. more likely to get stuck in local minima

Too small alpha means it will take "more" time to train; Perhaps too long.

Also, we don't have as much "bounce power" to find the best larger local minimas; We will likely be stuck in whatever "large" minima we started in. It's better to have a higher learning rate, then gradually decrease. Or at least start with a reasonable alpha.

Part 6 - Too Small # Neurons

If the # neurons are too small, then the accuracy will be low because the model under-fits the training set.

Part 7 - Too Small # Epochs

Too small a number of Epochs means our training may not converge as well as it could on the training data; Our model won't be able to predict as well or generalize as well. The fewer the epochs, the closer the model will behave toward random predictions. Oh well, at least we won't risk over-training.

Part 8 - Too Small Regularization Parameter (lambda)

Too small a regularization parameter might mean some neurons become saturated before others have a chance to learn correctly; Larger weight values will not be penalized as much as they should during gradient descent. This could mean our model fails to learn well. We have the potential to overfit the training data due to high variance.

Part 9 - Too Small Mini-Batch Size

Too-small mini-batch size means our training results will become closer to normal stochastic gradient descent (where we only pick one observation at a time). This would give poorer training results, as mini-batch is a good trade-off between memory consumption (consume "more" but not beyond the limits of our computer as with batch size equal to training set size) and more accurate results (we would get better training if we could compute the gradient over all observations at once).

Question 5 - Deep Net Training - Advices

You may have too many layers

- Too many layers may slow training, or cause vanishing/exploding gradient
- They may want to try fewer layers; More training examples; More epochs; More diverse training examples
- Once they implement these suggestions, the network may show more improvement for more epochs

They may be using too many layers and/or too many neurons

- Their model could have too-high variance, which means it's matching the training data too closely thus failing to improve on test data
- Try lowering the number of layers, or the number of neurons per layers
- With these suggestions, training time should improve, and perhaps the model will have higher bias and be able to generalize better

They may not have enough/diverse training data

- Deep neural networks require lots of training data to accurately model a problem; Usually more so than less-deep networks
- Try increasing the number of training samples, or perhaps generating fake training data using a GAN to produce realistic samples (if possible).
- With more training data, your model should show better ability to generalize correctly for the problem at hand.

They may not be using stochastic gradient descent / doing it incorrectly

- It is not wise to train the network by selecting training samples in the exact same order every time. If we cannot train against the entire dataset at once, we need to use either stochastic gradient descent or mini-batch.
- Try randomizing the order of the training set for each epoch.
- This will help your network train more generally with respect to the overall training set, and be less prone to getting stuck in local minima that deterministic/ordered may produce

Maybe Use Mini-Batch instead of Stochastic Gradient Descent

- Mini-batch is better than stochastic gradient because we're averaging the training over many samples at once rather than one sample. Perhaps the use of stochastic gradient descent is causing you to get struck in local minima
- Try using, like … mini-batch … and stuff
- This may help training to be more effective, as you're computing gradients somewhat more accurately by having more samples contribute to them at the same time.

They may not have architected the network well with respect to the actual problem

- They may be wasting computation time with a densely connected layer, and impeding the training, if the problem only needs "less" connections or a different topology or something
- Try analyzing the problem and seeing if a different topology or connection scheme may be appropriate; e.g., Image processing might be better used with convolutional layers followed by dense layers
- If they choose a better design, their network might improve and show better results

They may be using inappropriate activation functions

- Maybe ReLU is causing too many dead neurons, or sigmoid/tanh/etc are suffering from vanishing/exploding gradients.
- Find activations that are more appropriate for hidden layers, like perhaps leaky ReLU
- With better activations, you probably won't suffer from vanishing/exploding gradient, saturation, or dead neurons

They may have bad initial data

- Bad initial data could lead to bad local minima and other problems
- Try randomizing to NORMAL distribution using mean 0 and STD=(1/n), where n is the number of neurons... or try other schemes
- You may see your model finds better minima with better initialization

Improper use of regularization

- The regularization penalty could be too high and as such making it harder for the model to properly fit the data
- They can try lowering the regularization penalty and running the network with multiple different values for it to see which is the best
- They should notice that the network will better be able to fit their training data, what they need to be careful of is that they don't overfit the model by making the regularization penalty too low and as such making the model unable to generalize well.

Hitting Local Min or Max

- They hit a local max or min in the curve of the loss as such they are stuck there
- They should try starting again with new random weights and see if they can get past the point they are stuck on to see if it got past the local min they got stuck on previously, if that doesn't work you could try changing the hyper parameters such as learning rate so that it starts higher and then later shrinks after they have gotten past the point they were stuck on
- They should expect to see the loss go back up and then as they train and the new weights are adjusted they should see the Loss go back down and hopefully go past the point they were stuck at.