Christopher Tsai

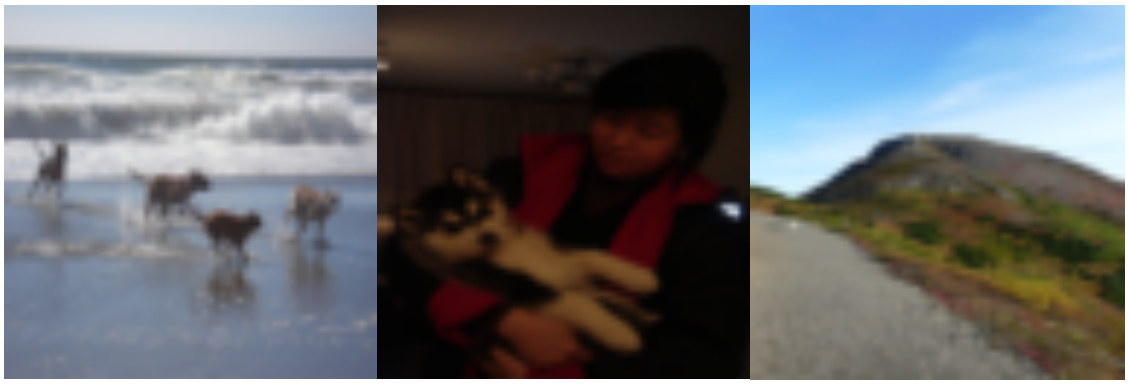
CS 349 HW #7 Free Response Questions

1. 

2. Training time (roughly) linearly increases as number of training examples increases, showcasing neural networks' main drawback: long training time. Slope of line is roughly 4 s per 500 examples, so for 60,000 examples (entire MNIST training dataset) it would take approximately solve(y – 4 = 4/500 * (60000 – 500), y) = 480 seconds = 0.1333 hrs. I am using 2020 MacBook Pro with 32 GB RAM.

3.

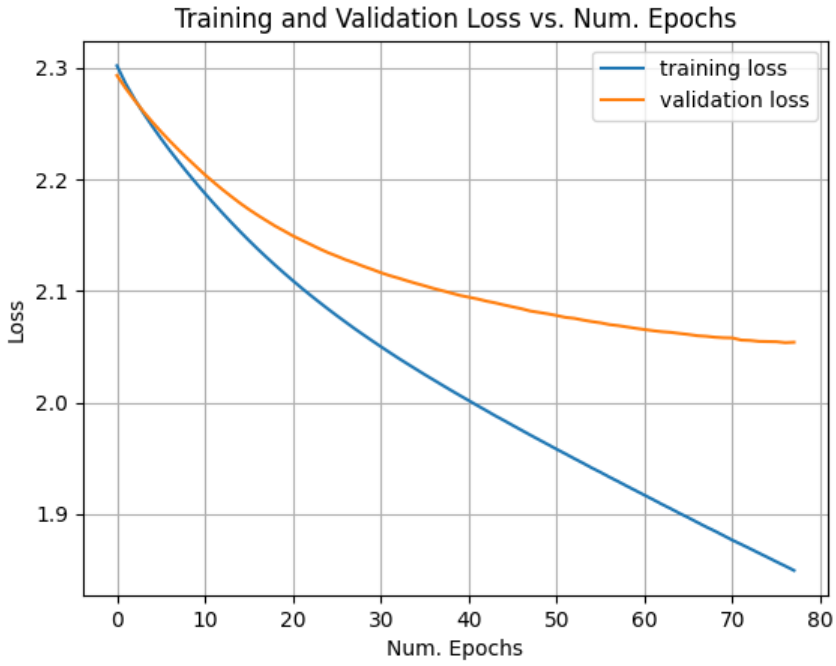
4. Testing accuracy (roughly) linearly increases as number of training examples increases. 2000 examples were enough to get over 90% of testing accuracy. The more the examples, the higher the training time and the better the testing accuracy.
5. The training set, validation set, and testing set have 7665, 2000, and 555 examples, respectively. The color palate of the images is RGB. There are 10 dog breeds to classify.
6. The following images (000009.png, 000011.png, and 000081.png) are all labelled with the breed of Siberian Husky.

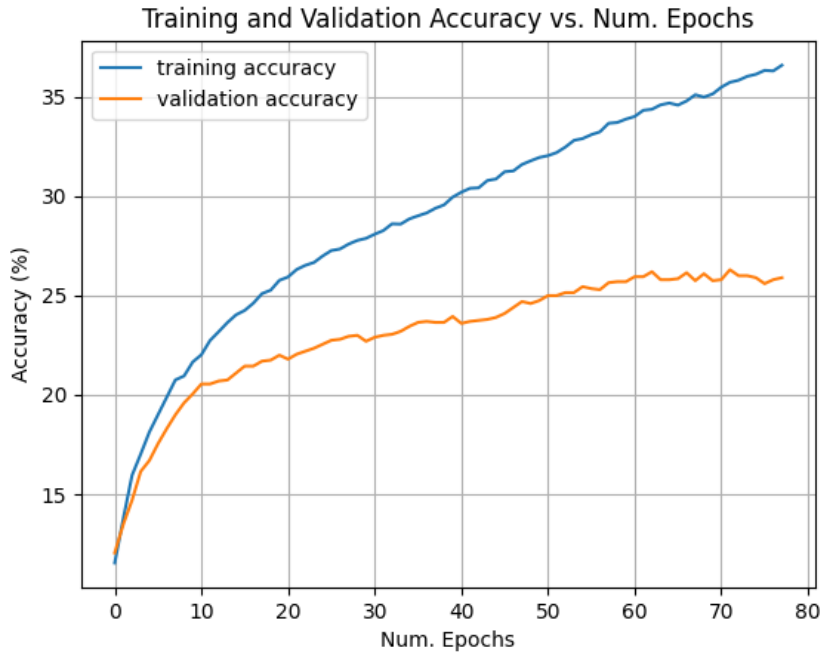


They are hard to classify, for different reasons. The first image contains multiple dogs and is taken from far away. There's barely any detail on how each dog looks like and even a human would struggle classifying them. The second image contains a person, which (as a living thing) could confuse a classifier. The dog in the second image is also a

puppy, which is harder to classify as most dogs aren't puppies. The third image is simply a landscape image. To the naked eye, it contains no dogs.

7. The network has 1581898 weights. Gotten from command "sum(param.numel() for param in model.parameters())".
8. Number of epochs before terminating = 78.





Accuracy on testing set = 23.24 %

9. We can see from the graphs that the model stopped training when validation results stopped getting better. This is synonymous to the validation accuracy no longer increasing and the validation loss no longer decreasing. We can see that the graphs stopped plotting as soon as this happens. The reason we stop at this point is that further training does not make the model learn properly, but rather overfitting occurs as the model gets used to the same training values (memorizes them, in a sense). If we hadn't stopped training, the validation loss would actually go up (and its accuracy would go down) with further training, which we don't want. The training loss would go down and reach a very low point, but this is due to overfitting.

15. It is possible to construct any neural network built using perceptrons to classify any binary function with binary inputs. For example, the Boolean operators "and", "or", and "not" can all be created using one perceptron each and the Boolean operator "xor" can be created by using a combination of perceptrons of the other Boolean operators. However, we're just talking "construct" here. If we want to learn the weights of such function using the perceptron training rule, cases are required to be linearly separable.

16. Linear activation functions have an advantage when compared to perceptrons because they minimize error even if examples are not linearly separable due to proper assignment of error (since they are differentiable). However, linear units only make linear decision surfaces so this method cannot be used to build nonlinear functions.

17. What we want is a targeted adversarial attack (for example, wanting an image of a 2 being recognized as a 5 by a neural network but clearly still a 2 to a human). To do this, we minimize a loss function that does not include W. We want to find an X' that is very similar to X but that gives a different Y (let's call it Y'). The loss function L(X,Y; X',Y;) then looks something like:

$$\frac{1}{2N}\sum_{i=i}^{N}\|Y' - Y_i\|_2^2 + \lambda\|X' - X_i\|_2^2$$

This is essentially minimizing squared loss, but for X' and Y' instead of W. λ is a hyperparameter that indicates how important minimizing the X difference is when compared to the Y difference.

I got some information from this article:

https://medium.com/@ml.at.berkeley/tricking-neural-networks-create-your-own-adversarial-examples-a61eb7620fd8