# MACHINE LEARNING PROJECT: PART 2

MNIST & CIFAR-10 CNN Exploration

DECEMBER 18, 2019

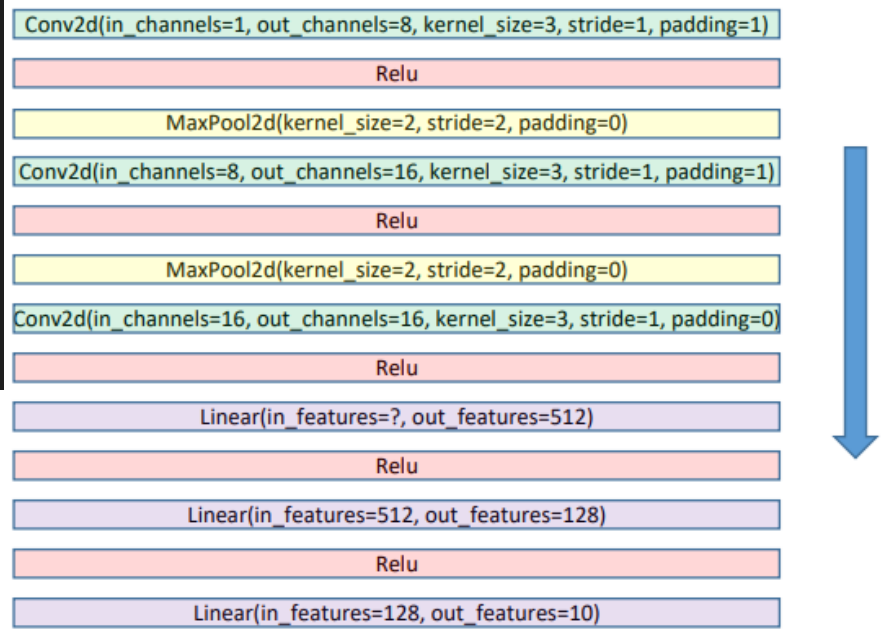
MARC SHEPHERD
Professor Jacob

**Part 1 – Baseline Network on MNIST Dataset**

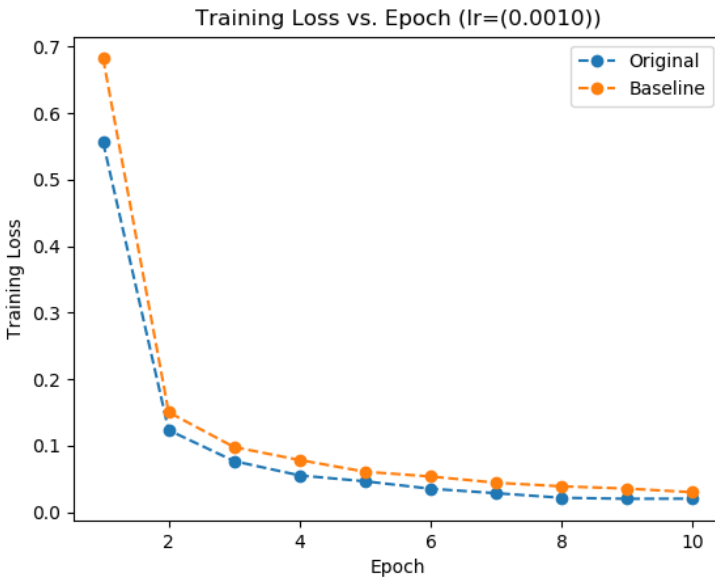
*1) Create the network as shown in figure 1*

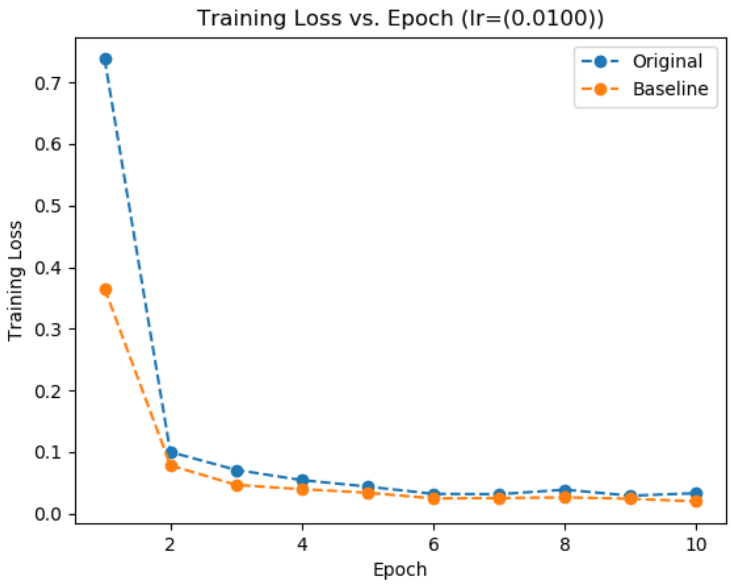
```python
def __init__(self):
    super(Net, self).__init__()
    self.conv1 = nn.Conv2d(1, 8, kernel_size=3, stride=1, padding=1)
    self.pool = nn.MaxPool2d(kernel_size=2, stride=2, padding=0)
    self.conv2 = nn.Conv2d(8, 16, kernel_size=3, stride=1, padding=1)
    self.conv3 = nn.Conv2d(16, 16, kernel_size=3, stride=1, padding=1)
    self.fc1 = nn.Linear(16 * 7 * 7, 512)
    self.fc2 = nn.Linear(512, 128)
    self.fc3 = nn.Linear(128, 10)

def forward(self, x):
    x = self.pool(F.relu(self.conv1(x)))
    x = self.pool(F.relu(self.conv2(x)))
    x = F.relu(self.conv3(x))
    x = x.view(-1, 16 * 7 * 7)
    x = F.relu(self.fc1(x))
    x = F.relu(self.fc2(x))
    x = self.fc3(x)
    return x
```




*2) Find a proper learning rate and plot the training loss vs. epoch*



*3) Compare the test performance with the baseline network provided to you on the MNIST dataset*

Exploring part 2; specifically, the learning rate section, lead me to a solid learning rate of 0.01. I also explored others, but for this first portion, I only compared 0.01 and 0.001, two of the best performing as seen in part 2. The comparisons are shown above in the plots.

```
Epoch:  0  validation loss:  0.170
Epoch:  1  train loss:  0.123
Epoch:  1  validation loss:  0.097
Epoch:  2  train loss:  0.077
Epoch:  2  validation loss:  0.076
Epoch:  3  train loss:  0.055
Epoch:  3  validation loss:  0.058
Epoch:  4  train loss:  0.046
Epoch:  4  validation loss:  0.064
Epoch:  5  train loss:  0.035
Epoch:  5  validation loss:  0.047
Epoch:  6  train loss:  0.029
Epoch:  6  validation loss:  0.042
Epoch:  7  train loss:  0.022
Epoch:  7  validation loss:  0.043
Epoch:  8  train loss:  0.020
Epoch:  8  validation loss:  0.050
Epoch:  9  train loss:  0.020
Epoch:  9  validation loss:  0.042
Finished Training in 6 mins
Accuracy of the network on the test images: 98.761 %
```

*Figure 3: Original - LR=0.001*

```
Epoch:  0  train loss:  0.682
Epoch:  0  validation loss:  0.194
Epoch:  1  train loss:  0.150
Epoch:  1  validation loss:  0.121
Epoch:  2  train loss:  0.098
Epoch:  2  validation loss:  0.087
Epoch:  3  train loss:  0.078
Epoch:  3  validation loss:  0.075
Epoch:  4  train loss:  0.061
Epoch:  4  validation loss:  0.063
Epoch:  5  train loss:  0.054
Epoch:  5  validation loss:  0.071
Epoch:  6  train loss:  0.044
Epoch:  6  validation loss:  0.054
Epoch:  7  train loss:  0.039
Epoch:  7  validation loss:  0.048
Epoch:  8  train loss:  0.036
Epoch:  8  validation loss:  0.044
Epoch:  9  train loss:  0.030
Epoch:  9  validation loss:  0.048
Finished Training in 4 mins
Accuracy of the network on the test images: 98.544 %
```

*Figure 4: Baseline - LR=0.001*

```
Epoch:  0  train loss:  0.738
Epoch:  0  validation loss:  0.134
Epoch:  1  train loss:  0.100
Epoch:  1  validation loss:  0.082
Epoch:  2  train loss:  0.071
Epoch:  2  validation loss:  0.079
Epoch:  3  train loss:  0.054
Epoch:  3  validation loss:  0.066
Epoch:  4  train loss:  0.044
Epoch:  4  validation loss:  0.067
Epoch:  5  train loss:  0.032
Epoch:  5  validation loss:  0.067
Epoch:  6  train loss:  0.031
Epoch:  6  validation loss:  0.086
Epoch:  7  train loss:  0.039
Epoch:  7  validation loss:  0.071
Epoch:  8  train loss:  0.029
Epoch:  8  validation loss:  0.082
Epoch:  9  train loss:  0.033
Epoch:  9  validation loss:  0.083
Finished Training in 6 mins
Accuracy of the network on the test images: 97.628 %
```

*Figure 1: Original - LR=0.01*

```
Epoch:  0  train loss:  0.365
Epoch:  0  validation loss:  0.109
Epoch:  1  train loss:  0.078
Epoch:  1  validation loss:  0.064
Epoch:  2  train loss:  0.047
Epoch:  2  validation loss:  0.065
Epoch:  3  train loss:  0.040
Epoch:  3  validation loss:  0.056
Epoch:  4  train loss:  0.034
Epoch:  4  validation loss:  0.050
Epoch:  5  train loss:  0.025
Epoch:  5  validation loss:  0.062
Epoch:  6  train loss:  0.025
Epoch:  6  validation loss:  0.058
Epoch:  7  train loss:  0.026
Epoch:  7  validation loss:  0.074
Epoch:  8  train loss:  0.024
Epoch:  8  validation loss:  0.078
Epoch:  9  train loss:  0.020
Epoch:  9  validation loss:  0.069
Finished Training in 3 mins
Accuracy of the network on the test images: 98.522 %
```

*Figure 2: Baseline - LR=0.01 (ran on google colab)*

You can see in the above results that both baseline runs had a significantly quicker runtime, >=2 minutes quicker. Neither CNN necessarily performed better given my restricted testing, but you can see the original had the high of 98.761%, and low of 97.628% on the validation set, whereas baseline had a greater average overall. In this case, it also seems that a learning rate of 0.001 performs better overall.

**Part 2 – Model Exploration**

**Part 2.1.1 (a)**

*The goal of this section is to understand the impact of the following hyperparameters and algorithmic choices on the performance of the system.*

- Learning Rate (LR) & Optimizer: Adam or SGD
  - Notes on readings:
    - Majority of learning rates fail to train the specified model
    - LR's too low never progress, and too high causes instability/no convergence
    - Training time can be greatly affected by learning rate
    - Hyper parameters are not invalidated by linear scaling the model
    - Adam is essentially a combo of RMSprop and Stochastic Gradient Descent w/ m
    - Adam is closing in on SGD w/ momentum to become the best optimization algo.
    - As referenced in a data science article, they mention a paper shows the optimal value for weight decay depends on number of iterations during training.
    - Though adaptive optimizers have better training performance, it doesn't imply higher accuracy, or, better generalization in valid data
    - In general, adam has the lowest training error & loss, but not validation

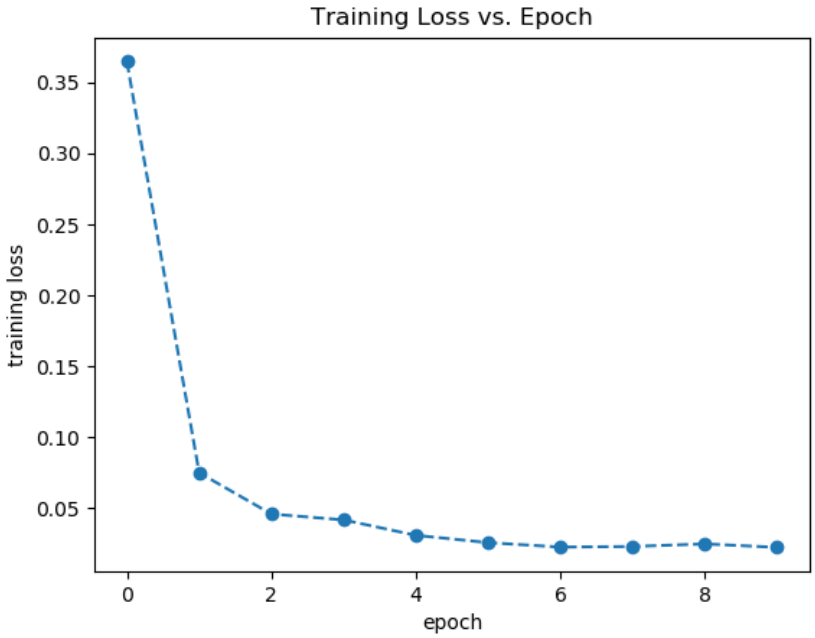
---

**Part 2.1.2 (b)**

- *Find proper LR for Adam and SGD, plot training loss vs training epoch number, and compare the convergence speed of the two optimizers and their respective test classification accuracies*
  - Adam shown below…

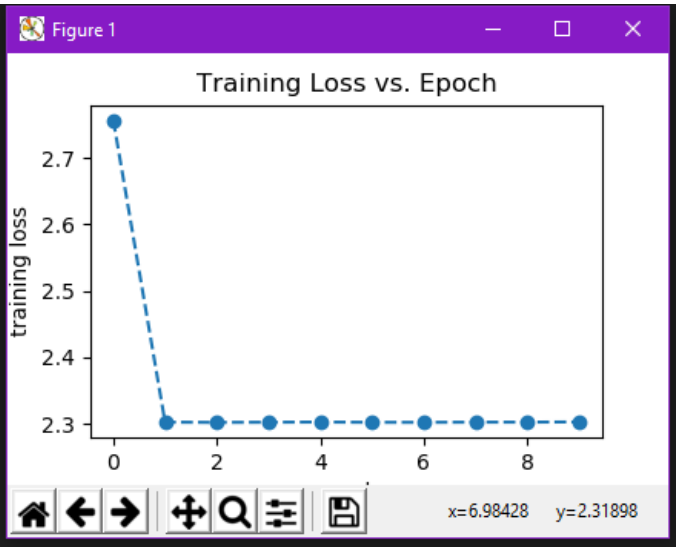
- Learning Rate of 0.01 - **good**

```
Epoch:  0  train loss:  0.365
Epoch:  0  validation loss:  0.115
Epoch:  1  train loss:  0.075
Epoch:  1  validation loss:  0.061
Epoch:  2  train loss:  0.046
Epoch:  2  validation loss:  0.060
Epoch:  3  train loss:  0.042
Epoch:  3  validation loss:  0.054
Epoch:  4  train loss:  0.031
Epoch:  4  validation loss:  0.054
Epoch:  5  train loss:  0.026
Epoch:  5  validation loss:  0.053
Epoch:  6  train loss:  0.023
Epoch:  6  validation loss:  0.062
Epoch:  7  train loss:  0.023
Epoch:  7  validation loss:  0.056
Epoch:  8  train loss:  0.025
Epoch:  8  validation loss:  0.068
Epoch:  9  train loss:  0.022
Epoch:  9  validation loss:  0.064
Finished Training in 4 mins
Accuracy of the network on the test images: 98.656 %
```



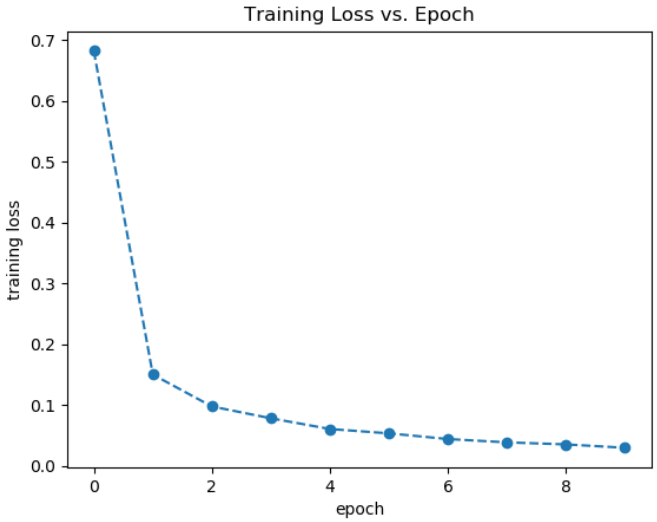
- Learning Rate of 0.05… you can see a bigger learning rate had hugely negative, or **poor**, results with regards to both loss and accuracy. Also, technically it converges, but the training loss is huge.

```
Epoch:  0  validation loss:  2.304
Epoch:  1  train loss:  2.303
Epoch:  1  validation loss:  2.302
Epoch:  2  train loss:  2.302
Epoch:  2  validation loss:  2.302
Epoch:  3  train loss:  2.302
Epoch:  3  validation loss:  2.302
Epoch:  4  train loss:  2.303
Epoch:  4  validation loss:  2.302
Epoch:  5  train loss:  2.302
Epoch:  5  validation loss:  2.303
Epoch:  6  train loss:  2.302
Epoch:  6  validation loss:  2.303
Epoch:  7  train loss:  2.303
Epoch:  7  validation loss:  2.304
Epoch:  8  train loss:  2.303
Epoch:  8  validation loss:  2.303
Epoch:  9  train loss:  2.303
Epoch:  9  validation loss:  2.302
Finished Training in 4 mins
Accuracy of the network on the test images: 11.250 %
```



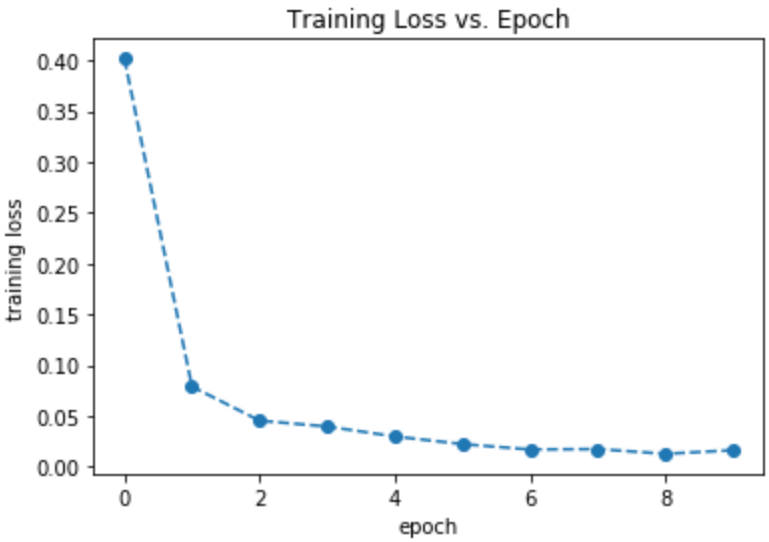

- Learning rate of 0.001 – still worse than 0.01, but ever so slightly, and only ran with one random seed value, so they are likely very similar -- **good**.

```
Epoch:  0  train loss:  0.682
Epoch:  0  validation loss:  0.194
Epoch:  1  train loss:  0.150
Epoch:  1  validation loss:  0.121
Epoch:  2  train loss:  0.098
Epoch:  2  validation loss:  0.087
Epoch:  3  train loss:  0.078
Epoch:  3  validation loss:  0.075
Epoch:  4  train loss:  0.061
Epoch:  4  validation loss:  0.063
Epoch:  5  train loss:  0.054
Epoch:  5  validation loss:  0.071
Epoch:  6  train loss:  0.044
Epoch:  6  validation loss:  0.054
Epoch:  7  train loss:  0.039
Epoch:  7  validation loss:  0.048
Epoch:  8  train loss:  0.036
Epoch:  8  validation loss:  0.044
Epoch:  9  train loss:  0.030
Epoch:  9  validation loss:  0.048
Finished Training in 4 mins
Accuracy of the network on the test images: 98.544 %
```

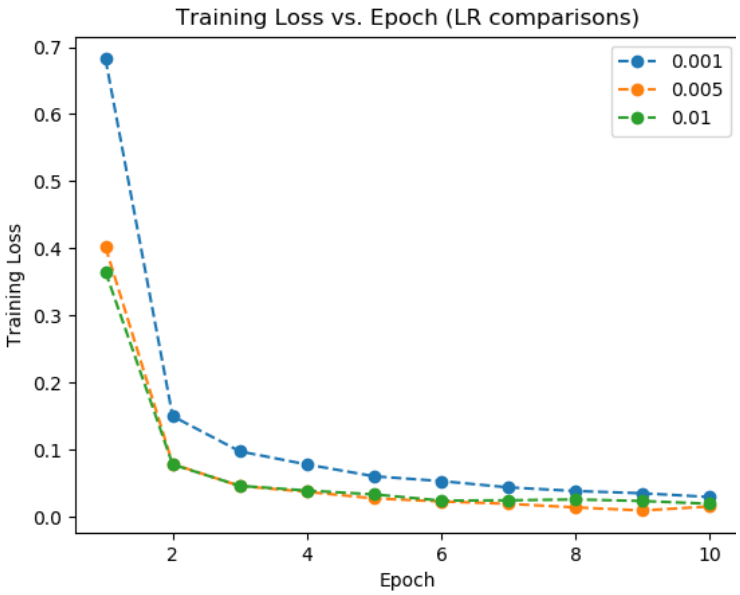


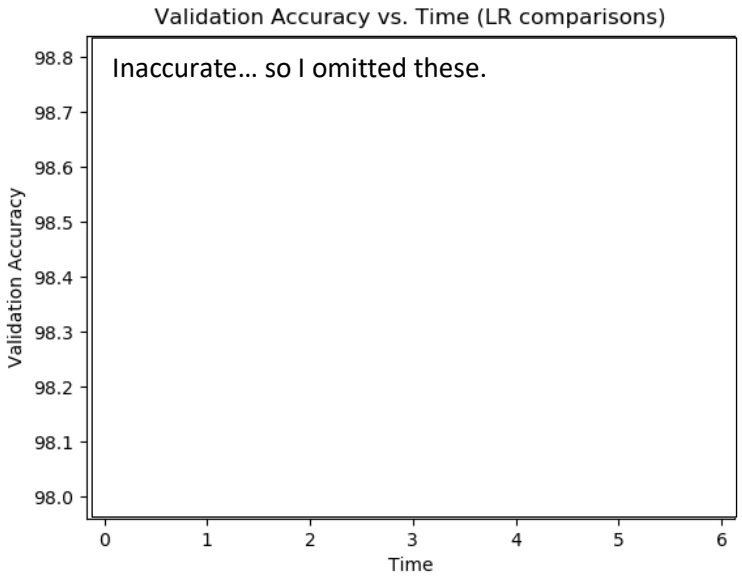
- Learning Rate of 0.005 – **great**

```
Epoch:  0  train loss:  0.402
Epoch:  0  validation loss:  0.101
Epoch:  1  train loss:  0.079
Epoch:  1  validation loss:  0.075
Epoch:  2  train loss:  0.045
Epoch:  2  validation loss:  0.049
Epoch:  3  train loss:  0.040
Epoch:  3  validation loss:  0.052
Epoch:  4  train loss:  0.030
Epoch:  4  validation loss:  0.055
Epoch:  5  train loss:  0.022
Epoch:  5  validation loss:  0.050
Epoch:  6  train loss:  0.017
Epoch:  6  validation loss:  0.058
Epoch:  7  train loss:  0.017
Epoch:  7  validation loss:  0.048
Epoch:  8  train loss:  0.013
Epoch:  8  validation loss:  0.053
Epoch:  9  train loss:  0.016
Epoch:  9  validation loss:  0.049
Finished Training in 3 mins
Accuracy of the network on the test images: 98.767 %
```

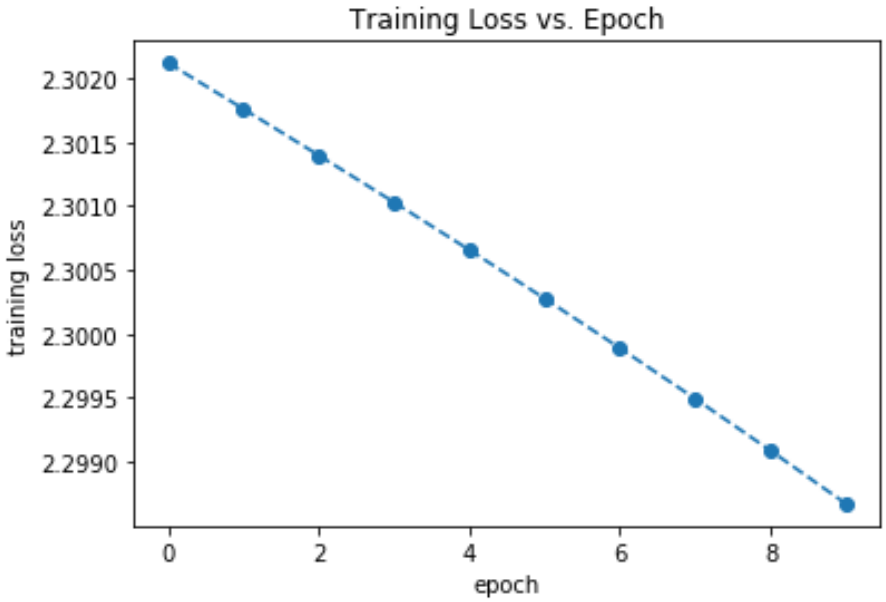



- Adam Comparisons – below you can see the comparisons between different Adam optimizer learning rates.



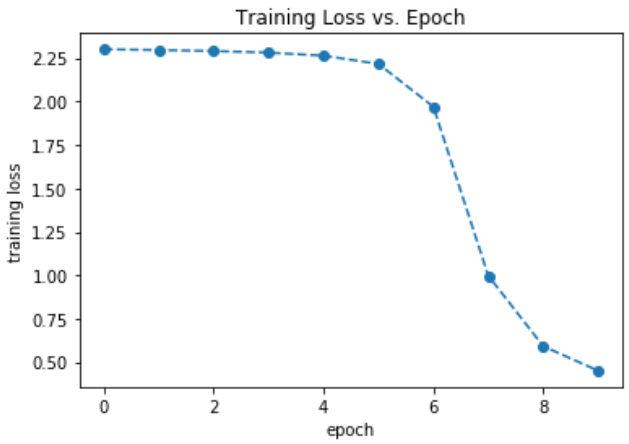



- SGD below…
  - Learning Rate of 0.001 -- **poor**



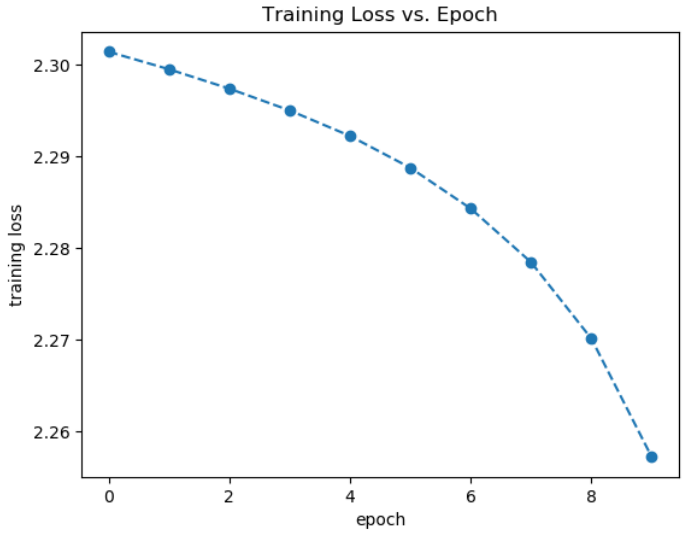

```
Epoch:  0  train loss:  2.302
Epoch:  0  validation loss:  2.302
Epoch:  1  train loss:  2.302
Epoch:  1  validation loss:  2.302
Epoch:  2  train loss:  2.301
Epoch:  2  validation loss:  2.301
Epoch:  3  train loss:  2.301
Epoch:  3  validation loss:  2.301
Epoch:  4  train loss:  2.301
Epoch:  4  validation loss:  2.301
Epoch:  5  train loss:  2.300
Epoch:  5  validation loss:  2.300
Epoch:  6  train loss:  2.300
Epoch:  6  validation loss:  2.300
Epoch:  7  train loss:  2.299
Epoch:  7  validation loss:  2.299
Epoch:  8  train loss:  2.299
Epoch:  8  validation loss:  2.299
Epoch:  9  train loss:  2.299
Epoch:  9  validation loss:  2.299
Finished Training in 4 mins
Accuracy of the network on the test images: 9.650 %
```

  - Learning Rate of 0.01 -- **ok**



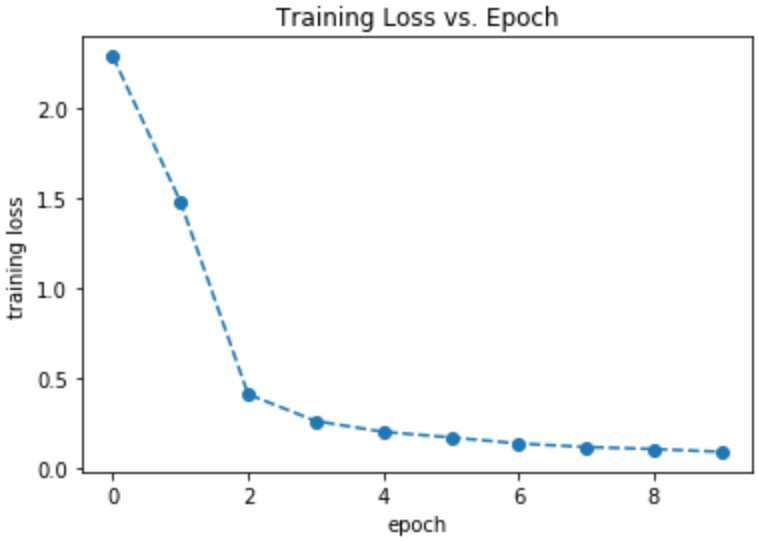

```
Epoch:  0  train loss:  2.300
Epoch:  0  validation loss:  2.299
Epoch:  1  train loss:  2.296
Epoch:  1  validation loss:  2.294
Epoch:  2  train loss:  2.291
Epoch:  2  validation loss:  2.287
Epoch:  3  train loss:  2.282
Epoch:  3  validation loss:  2.275
Epoch:  4  train loss:  2.264
Epoch:  4  validation loss:  2.249
Epoch:  5  train loss:  2.216
Epoch:  5  validation loss:  2.164
Epoch:  6  train loss:  1.970
Epoch:  6  validation loss:  1.569
Epoch:  7  train loss:  0.997
Epoch:  7  validation loss:  0.674
Epoch:  8  train loss:  0.592
Epoch:  8  validation loss:  0.522
Epoch:  9  train loss:  0.451
Epoch:  9  validation loss:  0.459
Finished Training in 4 mins
Accuracy of the network on the test images: 85.283 %
```

- Learning Rate of 0.005 – **poor**, maybe with higher iterations it would perform better



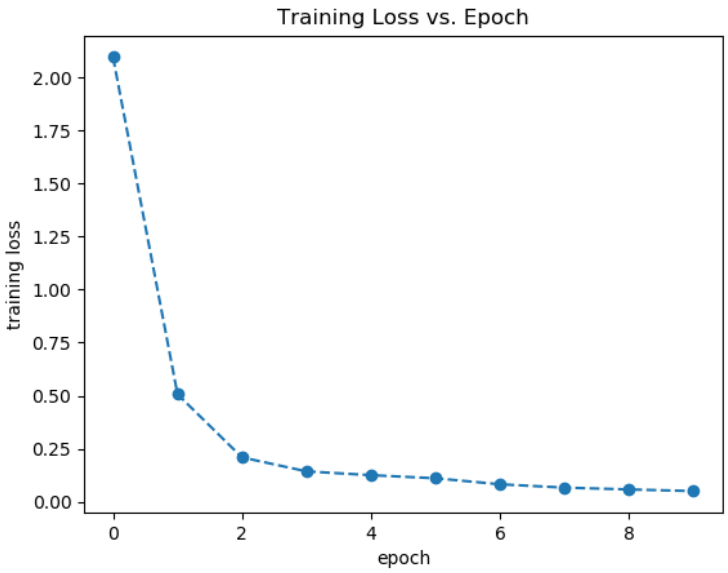

```
Epoch:  0  train loss:  2.301
Epoch:  0  validation loss:  2.301
Epoch:  1  train loss:  2.300
Epoch:  1  validation loss:  2.299
Epoch:  2  train loss:  2.297
Epoch:  2  validation loss:  2.296
Epoch:  3  train loss:  2.295
Epoch:  3  validation loss:  2.294
Epoch:  4  train loss:  2.292
Epoch:  4  validation loss:  2.291
Epoch:  5  train loss:  2.289
Epoch:  5  validation loss:  2.287
Epoch:  6  train loss:  2.284
Epoch:  6  validation loss:  2.282
Epoch:  7  train loss:  2.278
Epoch:  7  validation loss:  2.275
Epoch:  8  train loss:  2.270
Epoch:  8  validation loss:  2.265
Epoch:  9  train loss:  2.257
Epoch:  9  validation loss:  2.249
Finished Training in 4 mins
Accuracy of the network on the test images: 51.750 %
```

- Learning Rate of 0.05 -- **good**



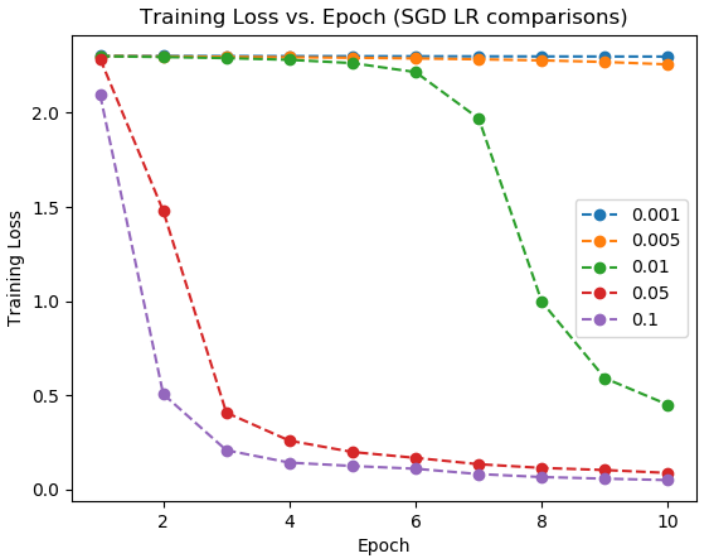

```
Epoch:  0  train loss:  2.287
Epoch:  0  validation loss:  2.251
Epoch:  1  train loss:  1.479
Epoch:  1  validation loss:  0.680
Epoch:  2  train loss:  0.409
Epoch:  2  validation loss:  0.308
Epoch:  3  train loss:  0.260
Epoch:  3  validation loss:  0.221
Epoch:  4  train loss:  0.200
Epoch:  4  validation loss:  0.176
Epoch:  5  train loss:  0.169
Epoch:  5  validation loss:  0.147
Epoch:  6  train loss:  0.135
Epoch:  6  validation loss:  0.139
Epoch:  7  train loss:  0.115
Epoch:  7  validation loss:  0.113
Epoch:  8  train loss:  0.104
Epoch:  8  validation loss:  0.110
Epoch:  9  train loss:  0.089
Epoch:  9  validation loss:  0.107
Finished Training in 3 mins
Accuracy of the network on the test images: 96.678 %
```

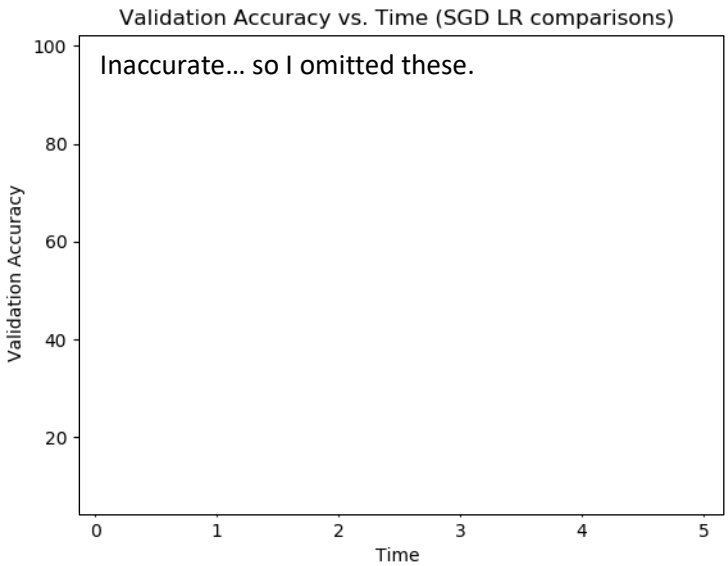
- Learning Rate of 0.1 -- **good**

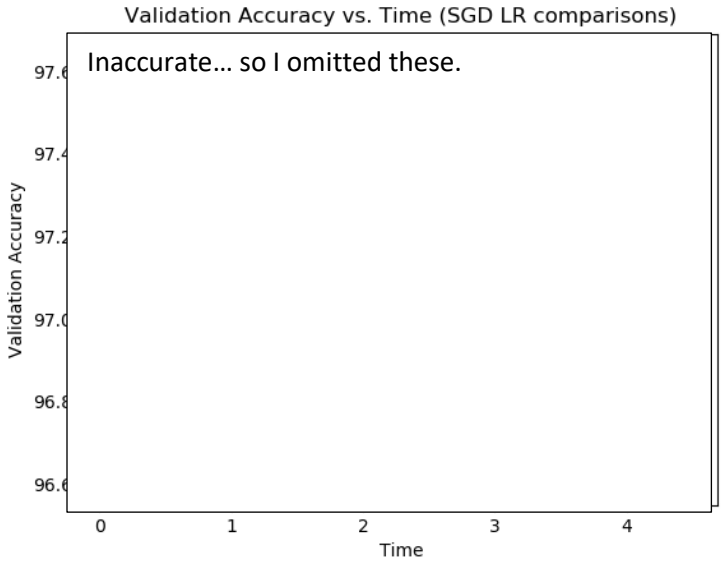



```
Epoch:  0  train loss:  2.096
Epoch:  0  validation loss:  1.056
Epoch:  1  train loss:  0.507
Epoch:  1  validation loss:  0.257
Epoch:  2  train loss:  0.209
Epoch:  2  validation loss:  0.187
Epoch:  3  train loss:  0.144
Epoch:  3  validation loss:  0.159
Epoch:  4  train loss:  0.126
Epoch:  4  validation loss:  0.108
Epoch:  5  train loss:  0.111
Epoch:  5  validation loss:  0.225
Epoch:  6  train loss:  0.083
Epoch:  6  validation loss:  0.085
Epoch:  7  train loss:  0.067
Epoch:  7  validation loss:  0.076
Epoch:  8  train loss:  0.059
Epoch:  8  validation loss:  0.075
Epoch:  9  train loss:  0.051
Epoch:  9  validation loss:  0.076
Finished Training in 4 mins
Accuracy of the network on the test images: 97.556 %
```

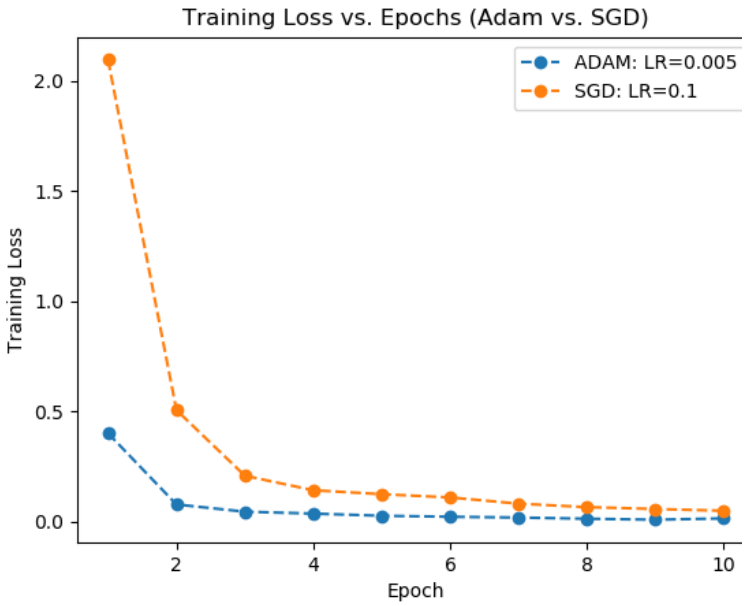
- SGD Comparisons – below you can see the comparisons between different SGD optimizer learning rates.

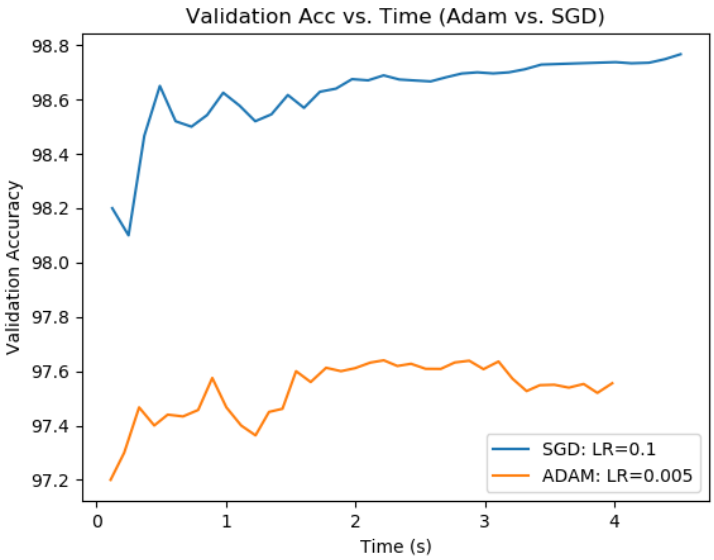





- After laborious analyses, I found the proper/best learning rate for both optimizers: Adam and SGD. Though random number seeds lead me to be wary within 0.5% or so, I decided to go ahead and use the highest working accuracy on the validation set. **Adam** came out to hit 98.767%, which held a learning rate of **0.005**. **SGD**, on the other hand, hit 97.556%, which held a learning rate of **0.1.**



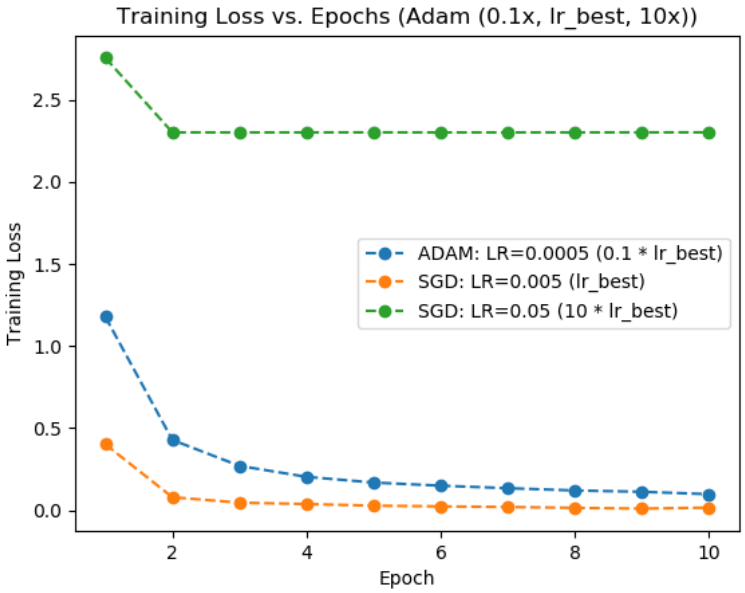
**Part 2.1.3 (c)**

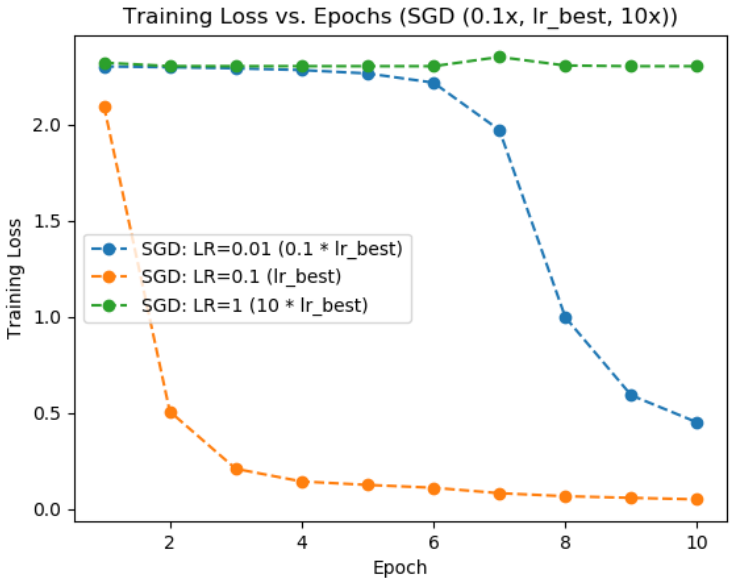
*Describe the lessons you learn from the experiments. Specifically compare the training convergence for the three learning rates (0.1 x lr_best, lr_best and 10 x lr_best).*

Given lr_best_adam = 0.005, lr_best_SGD = 0.1…

For adam, try $0.1 * 0.005 = 0.0005$, 0.005, and $10 * 0.005 = 0.05$

For SGD, try $0.1 * 0.1 = 0.01$, 0.1, and $10 * 0.1 = 1$




Recall from the notes…

LR's too low never progress, and too high causes instability/no convergence

This is the main general trend I have learned doing these experiments. Higher learning rates tend to not converge (or 'converge' with an obscene loss and stay that way). The interesting run to me is SGD with a learning rate of 0.01 (the blue line), where you can see it is starting to converge, but it doesn't start converging with importance until around the 8th or 9th epoch, whereas the lr_best of 0.1 converges on just the 2nd to 3rd epoch. My general takeaway from this is to trend, or start, at lower learning rates and slowly increase your estimate until you get solid performance.

***Part 2.2***

**2.2.1 (a)**

- ❖ Activation Functions
  - ➢ Notes on readings:
    - Sigmoid function saturate, and are only sensitive to changes around the input midpoint
      - It becomes hard to adapt the weight/improve performance after saturated
    - Deep CNN's ~~are~~ were difficult for sigmoid (vanishing gradient problem)
      - ~~Though ReLU can in some cases fix the vanishing gradient problem~~
      - ReLU DOES fix the vanishing gradient problem in most cases
    - ReLU
      - Default activation function for CNN and MP
      - Nonlinear activation function… but behaves like a linear one
      - has become the default activation function for most CNN
      - is trivial to implement, whereas sigmoid uses exponential calculation
      - capable of outputting a true zero value → can lead to acceleration of learning
      - A limitation is where large weight updates lead to the summed input being (-) always
        - ♦ See Leaky ReLU (LRel), ELU, PReLU, for a patch/fix to this issue
    - …consider setting bias to a small value (e.g. 0.1)
    - Weights of a NN must be initialized to small random values
      - Modification to Xavier initialization → He initialization +/-sqrt(2/n)
    - Good practice to scale input data (e.g. standardizing variables to have…
      - zero mean
      - Unit variance / normalizing each value to a 0 – 1 scale
    - May be a good idea to use a form of weight regularization, L1/L2 vector norm

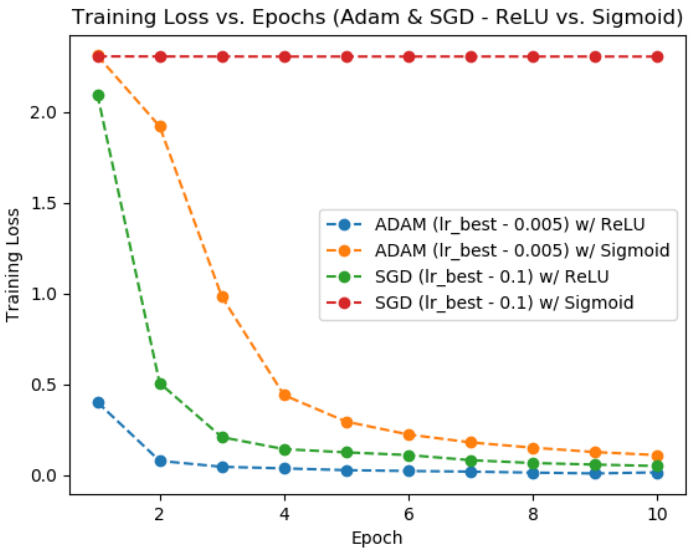
**2.2.2 (b)**

*Train two networks with Sigmoid and Relu as respective activation functions* – below…

**2.2.3 (c)**

*Test and compare the training convergence speeds and classification accuracies on the test dataset. Give your observation*

You can see from the graph I plotted below with of the different optimizers (adam v. SGD) each trained with ReLU and sigmoid. Like I assumed, ReLU performed better for both optimizers, and even converged quicker. You can see with SGD, sigmoid acting as the activation function failed to even get the SGD optimizer's training loss down (failed to converge properly).




```
Accuracy of the network on the test images:
```

*SGD*

*Sigmoid – 10.794% | ReLU – 97.556 %*

*Adam*

*Sigmoid – 95.822 % | ReLU – 98.767 %*

You can see the same pattern here with the accuracy of the network on the test images, where the adam optimizer outperforms in general, but also, more importantly, that ReLU outperforms in both circumstances.

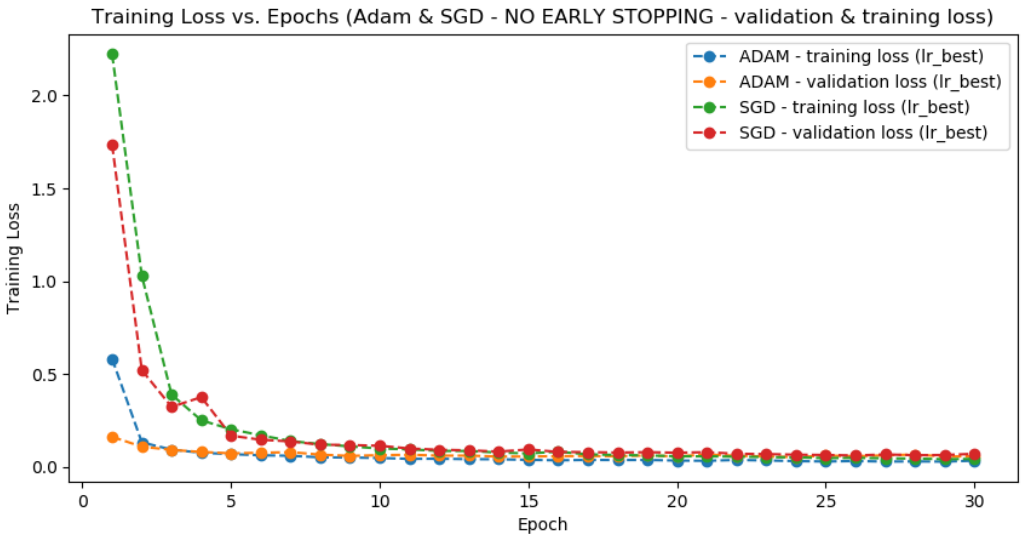
## Part 2.3 – Early Stopping

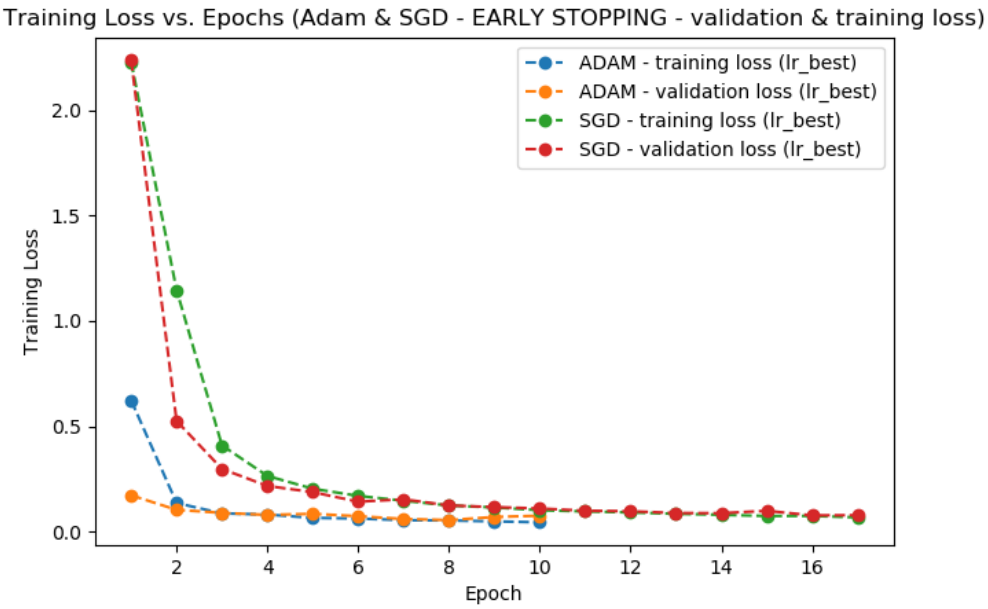
### Part 2.3.1 (a)

- Early Stopping
  - Little training leads to underfitting of training and test sets
  - Too much training leads to overfitting of training, and poor performance on test set
  - SOLUTION → train and stop when performance on validation dataset starts to degrade (ES)
  - Simple case → training stopped when validation test set decreases (i.e. increase in loss)
    - In practice, however, fluctuations are common, so be careful…
      - Generally, 'slower' stops lead to improved generalization

I decided to implement a simple early stopping strategy. I simply gave a slight leniency for the validation loss degrading (being higher than the previous), which was carried out by allowing it 3 'mishaps'. On the third increase in loss, I 'early-stop'.

I trained the network (given 30 epochs) with the early stopping method in place. Below is the visualization of the training and validation loss vs. the training epochs.



### Part 2.3.2 (b) & 2.3.3 (c)

Remember, the model is underfitting if the accuracy on the validation set is higher than the accuracy on the training set (and also if the whole model performs badly). Alternatively, the model can overfit if the training accuracy is higher than the validation set accuracy.

With early stopping:

- Adam – training accuracy ➔ 97.989 %, validation ➔ 97.28 % (probably, good)
- SGD – training accuracy ➔ 97.394 %, validation ➔ 90.69 % (possibly overfit, or poor performance)

Without early stopping:

- Adam – training accuracy ➔ 98.411 %, validation ➔ 98.002 %
- SGD – training accuracy ➔ 97.878 %, validation ➔ 94.43 % (possibly overfit, or poor performance)

I was surprised to see little difference in the creation of the visualization difference between early stopping and no early stopping. I even increased the epoch count to 30 to see a difference, but there wasn't much of one. Maybe if I were to increase it to say, 100 epochs, there would be, but that would take an eternity. It was cool to see, however, that with early stopping, the training was cut short around the $13^{th}$ epoch on average. Without early stopping, validation loss hit 13 early stop thresholds for the SGD run, where the current validation loss was higher than the previous validation loss. This is a general trend towards overfitting, which is why early stopping is put in place. All in all, it looks like there is *potentially a slight overfitting problem, but it's not on the scale I was expecting* (thought it would be much more).

**Part 2.4 – Data Augmentation**

- Data Augmentation
  - Invariance – a CNN that can robustly classify objects even placed in different orientations
    - A CNN can be invariant to translation, viewpoint, size, illumination
  - Offline augmentation – good for small datasets
  - Online augmentation – good for large datasets
    - Transformations on mini-batches then feed to model
  - Use it smartly, not just to increase data (no need for irrelevant data)
  - For this project, using *torchvision.transforms* will allow augmentation on dataset

Below are the transforms I applied to augment the training data…

```
transform = transforms.Compose(
    [
    transforms.RandomRotation(degrees=40),
    transforms.ColorJitter(brightness=0.3, contrast=0.4, hue=0.5, saturation=0.8),
    transforms.ToTensor(),
    transforms.Normalize((0.1307,), (0.3081,))])
```

Without augmentation…

```
Finished Training in 6 mins
Accuracy of the network on the test images: 98.011 %
```

With augmentation…

```
Finished Training in 9 mins
Accuracy of the network on the test images: 98.272 %
```

It's nice, because applying these augmentations to the training set dynamically transforms samples (random transformations) at every epoch, 'on the fly', making it great for training the network. A cool thing to note in these trainings, is that the 'without augmentation' training lead to 8 early stop counts, whereas the 'with augmentation' training lead to only 2 early stop counts, giving a bit of evidence that the augmentation did in fact help, specifically to curtail degradation of validation loss. Aside from that, you can see the accuracy is indeed better with augmentation. From reading articles, a 0.2 % difference is nothing to scoff at, however I am still not familiar enough to know if this is a significant increase. Nonetheless, performance is better with augmentation, which is what I assumed would occur, as the network is more adept at noticing differences because of it.

**Part 2.5 – Network Depth vs. Network Width**

- Network { Depth vs. Width }
  - Network depth is number of layers
  - Network width is the max number of nodes in a layer
    - Corresponds to different color channels of an image . . .?
    - Depth of a conv. Layer → number of filters
  - http://proceedings.mlr.press/v33/pandey14.pdf
  - https://medium.com/finc-engineering/cnn-do-we-need-to-go-deeper-afe1041e263e

5 Layer Network

```
# construct the CNN-----------------------------------------------------
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(1, 8, kernel_size=3, stride=1, padding=1)
        self.pool = nn.MaxPool2d(kernel_size=2, stride=2, padding=0)
        self.conv2 = nn.Conv2d(8, 16, kernel_size=3, stride=1, padding=1)
        self.conv3 = nn.Conv2d(16, 16, kernel_size=3, stride=1, padding=1)
        self.conv4 = nn.Conv2d(16, 16, kernel_size=3, stride=1, padding=1)
        self.conv5 = nn.Conv2d(16, 16, kernel_size=3, stride=1, padding=1)
        self.fc1 = nn.Linear(16 * 7 * 7, 512)
        self.fc2 = nn.Linear(512, 128)
        self.fc3 = nn.Linear(128, 10)

    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = F.relu(self.conv3(x))
        x = F.relu(self.conv4(x))
        x = F.relu(self.conv5(x))
        x = x.view(-1, 16 * 7 * 7)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x
```

5 layers - `Accuracy of the network on the test images: 98.094 %`

3 layers (from earlier) - `Accuracy of the network on the test images: 98.011 %`

The early stop count for the 5 layer network was 5, as compared to 8 (not super significant). Accuracy for the 5-layer network as compared to the 3 layer network was just slightly above, which I would not rule as significant, either. Nevertheless, the 5-layer network did perform slightly better, so I can rule it in this case that deeper is better.

Deeper, or more layers, allows the network to perform more convolutions, letting it extract features with more precision. Deeper networks have been more common nowadays because we have the computing power to do so. Deeper networks, especially VERY deep ones, with upwards of 1000 layers are likely to overfit. In summary, I believe deeper networks do help, but only to a certain extent. From researching other articles and papers, the consensus seems to be more layers helps, but tapers off after a certain point (usually in the single digits).

Note: Also, I did not pool or use softmax, which may have helped performance even more.

**PART 3**

*The goal of this component is to extend the above model to CIFAR 10 dataset and report the testing performance. Note that the CIFAR10 dataset requires more training time. It may be difficult to vary the parameters and test their impact. You may look at https://pytorch.org/tutorials/beginner/blitz/cifar10_tutorial.html for examples.*

```python
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(3, 6, kernel_size=5, stride=1, padding=1)
        self.pool = nn.MaxPool2d(kernel_size=2, stride=2, padding=0)
        self.conv2 = nn.Conv2d(6, 16, kernel_size=5, stride=1, padding=0)
        # self.conv3 = nn.Conv2d(16, 16, kernel_size=3, stride=1, padding=1)
        self.fc1 = nn.Linear(16 * 5 * 5, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)

    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        # x = F.relu(self.conv3(x))
        x = x.view(-1, 16 * 5 * 5)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x
```

I created a network similar to the one as shown in the beginner Cifar 60-minute blitz. Results below…

On 15 epochs with Adam & learning rate of 0.005…

```
Finished Training in 12 mins
Accuracy of the network on the test images: 9.873 %
```

On 6 epochs to get a feel for the learning rate (0.004)… there was a large improvement (still bad though)

```
Accuracy of the network on the test images: 29.260 %
```

Obviously, there is huge room for improvement. There is no data augmentation, early stopping, different choices for optimizers (can try SGD with momentum), and I only tested two learning rates. More exploring will be done below in the extra credit area!

**PART 4: EXTRA CREDIT (EXPERIMENTATION ON CIFAR-10 DATA)**

Early stop after 5 degradations of validation loss… (held constant for now)

Testing different learning rates with 4 epochs, then will expand to more epochs

Data augmentation ➔ color jitter is good for this dataset, rotation is so-so

Optimizer: Adam for now

Experimentation below…

Learning rate of 0.002 → `Accuracy of the network on the test images: 45.980 %`

```
Epoch:  10  validation loss:  1.231  validation accuracy:  54.32181818181818
Epoch:  11  train loss:  1.114
Early stop count 1
Epoch:  11  validation loss:  1.263  validation accuracy:  54.65
Epoch:  12  train loss:  1.101
Epoch:  12  validation loss:  1.242  validation accuracy:  54.93794871794872
Epoch:  13  train loss:  1.095
Early stop count 2
Epoch:  13  validation loss:  1.278  validation accuracy:  55.215714285714284
Epoch:  14  train loss:  1.087
Epoch:  14  validation loss:  1.252  validation accuracy:  55.370666666666665
Epoch:  15  train loss:  1.075
Epoch:  15  validation loss:  1.213  validation accuracy:  55.62125
Epoch:  16  train loss:  1.075
Epoch:  16  validation loss:  1.212  validation accuracy:  55.888627450980394
Epoch:  17  train loss:  1.053
Early stop count 3
Epoch:  17  validation loss:  1.217  validation accuracy:  56.120370370370374
Epoch:  18  train loss:  1.057
Early stop count 4
Epoch:  18  validation loss:  1.280  validation accuracy:  56.282105263157895
Epoch:  19  train loss:  1.046
Epoch:  19  validation loss:  1.218  validation accuracy:  56.437666666666665
Epoch:  20  train loss:  1.047
Epoch:  20  validation loss:  1.209  validation accuracy:  56.639047619047616
Epoch:  21  train loss:  1.044
Epoch:  21  validation loss:  1.205  validation accuracy:  56.77787878787879
Epoch:  22  train loss:  1.037
Early stop count 5
Epoch:  22  validation loss:  1.220  validation accuracy:  56.948115942028984
Epoch:  23  train loss:  1.038
Epoch:  23  validation loss:  1.208  validation accuracy:  57.07222222222222
Epoch:  24  train loss:  1.034
Epoch:  24  validation loss:  1.203  validation accuracy:  57.1896
Epoch:  25  train loss:  1.021
Early stop count 6
Epoch:  25  validation loss:  1.278  validation accuracy:  57.28897435897436
Epoch:  26  train loss:  1.026
Epoch:  26  validation loss:  1.232  validation accuracy:  57.410617283950614
Epoch:  27  train loss:  1.020
Epoch:  27  validation loss:  1.211  validation accuracy:  57.53285714285714
Epoch:  28  train loss:  1.016
Epoch:  28  validation loss:  1.205  validation accuracy:  57.6351724137931
Epoch:  29  train loss:  1.014
Early stop count 7
Epoch:  29  validation loss:  1.229  validation accuracy:  57.733777777777775
EARLY STOP ENGAGING AT EPOCH:  30
Finished Training in 34 mins
Accuracy of the network on the test images: 60.473 %
```

You can see below that I obtained an accuracy of 60.473 % for CIFAR-10. Obviously this is not amazing, but it's much better than my earlier attempts, and on my CPU, any bigger network would take much longer to complete.

**Used…**

Optimizer: Adam

Activation function: ReLU

```
learning_rate = 0.001
EPOCHS = 50
EARLY_STOP_THRESHOLD = 7
```

Learning rate of 0.001

Early Stopping: 50 epochs (early stopped at 30)

```
transforms.RandomHorizontalFlip(),
transforms.ColorJitter(brightness=0.1, contrast=0.5, hue=0.5, saturation=0.5),
```

Data augmentation used below…