Lab4 Writeup
Problem 1

1. Setup

We will start by fetching cifar_10_small using `fetch_openml` from scikitlearn. We can run the following script in order to get the dataset.

```
# Fetch the CIFAR-10-Small dataset
cifar_10_small = fetch_openml(name="CIFAR_10_small", version=1,parser='auto')

# Access the data and target labels
X, y = cifar_10_small.data, cifar_10_small.target

print(X.shape,y.shape)
# (20000, 3072) (20000,)
```

Sweet, so now we will go ahead and try to display one of the images. We can use the following script in order to display one of the images.

```
# Display one of the images

# Helper Function to Display the image
def display_image(image_data):
    image = image_data.reshape(3, 32, 32).transpose(1, 2, 0)

    plt.imshow(image)
    plt.axis('off')
    plt.show()

# Selecting which index to show
X = X.to_numpy()

display_image(X[1]) # Cool truck
```

Nice, and the final part of setup that we can do is to create a train test split for the data. We can do that by running the following script.

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.25, random_state=42)
```

2. Creating Model

We can go ahead a create a `LogisticRegression` model using scikitlearn. We will also use `cross_val_score` in order to run a cross validation on our splits. We can run the following script and we get.

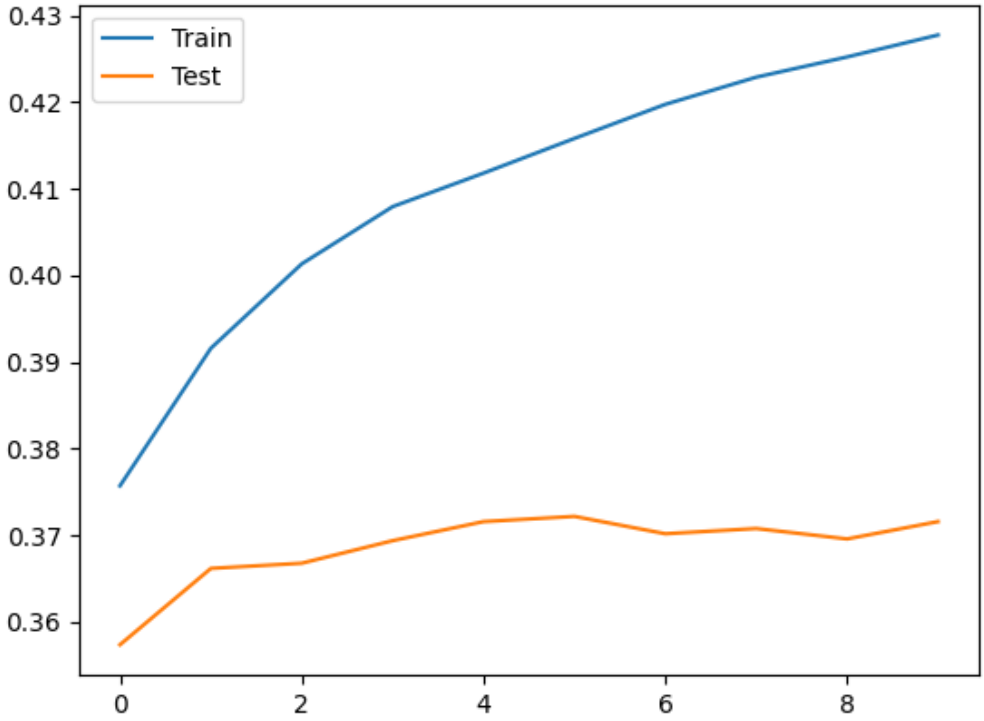
```
lg = LogisticRegression(multi_class='multinomial', penalty='elasticnet', C=1, solver='saga',
verbose=1, n_jobs=-1, l1_ratio=.5, random_state=42)

scores = cross_val_score(lg,X,y,cv=4)

print(scores.mean())
```

After running for about 30 minutes we get a mean accuracy of `0.3777` for our cross validation, which similar to other `LogisticRegression` models on CIFAR-10. We can use the script below to show the training and validation accuracy.

```
# Create Model
lg = LogisticRegression(multi_class='multinomial',penalty='elasticnet',C=1,solver='saga', verbose=1,
n_jobs=-1, l1_ratio=.5,max_iter=100,random_state=42)
lg.fit(X_train,y_train)

# Predict probabilities for training and test data
train_probabilities = lg.predict_proba(X_train)
test_probabilities = lg.predict_proba(X_test)

# Calculate log loss for training and test data
train_log_loss = log_loss(y_train, train_probabilities)
test_log_loss = log_loss(y_test, test_probabilities)
```

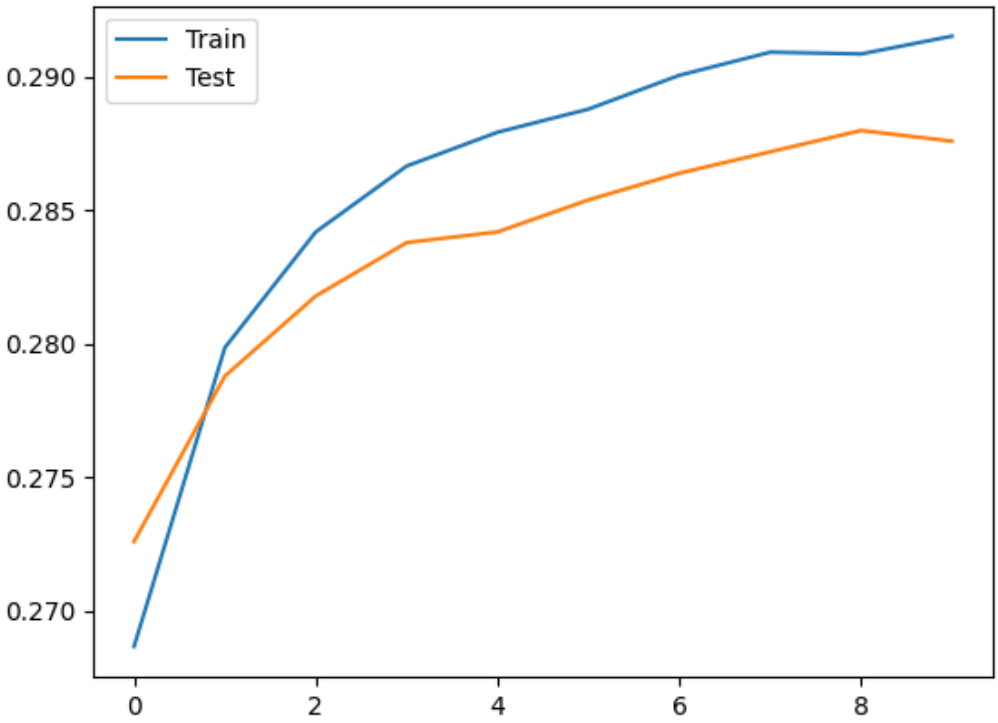
We get a `training_loss` of 1.44320 and a `test_loss` of 1.81275. We can take it one step further and plot the training and test loss over each epoch with the following script.

```
lg = LogisticRegression(multi_class='multinomial',penalty='elasticnet',C=1,solver='saga', verbose=1,
n_jobs=-1, l1_ratio=.5,max_iter=1,warm_start=True,random_state=42)

epochs = 10
training_loss = []
validation_loss = []
for epoch in range(epochs):
    lg = lg.fit(X_train, y_train)
    Y_pred = lg.predict(X_train)
    curr_train_score = accuracy_score(y_train, Y_pred)   # training performances
    Y_pred = lg.predict(X_test)
    curr_valid_score = accuracy_score(y_test, Y_pred)    # validation performances
    training_loss.append(curr_train_score)               # list of training perf to plot
    validation_loss.append(curr_valid_score)             # list of valid perf to plot

plt.plot(range(epochs),training_loss,label='Train')
plt.plot(range(epochs),validation_loss,label='Test')
plt.legend()
plt.show()
```

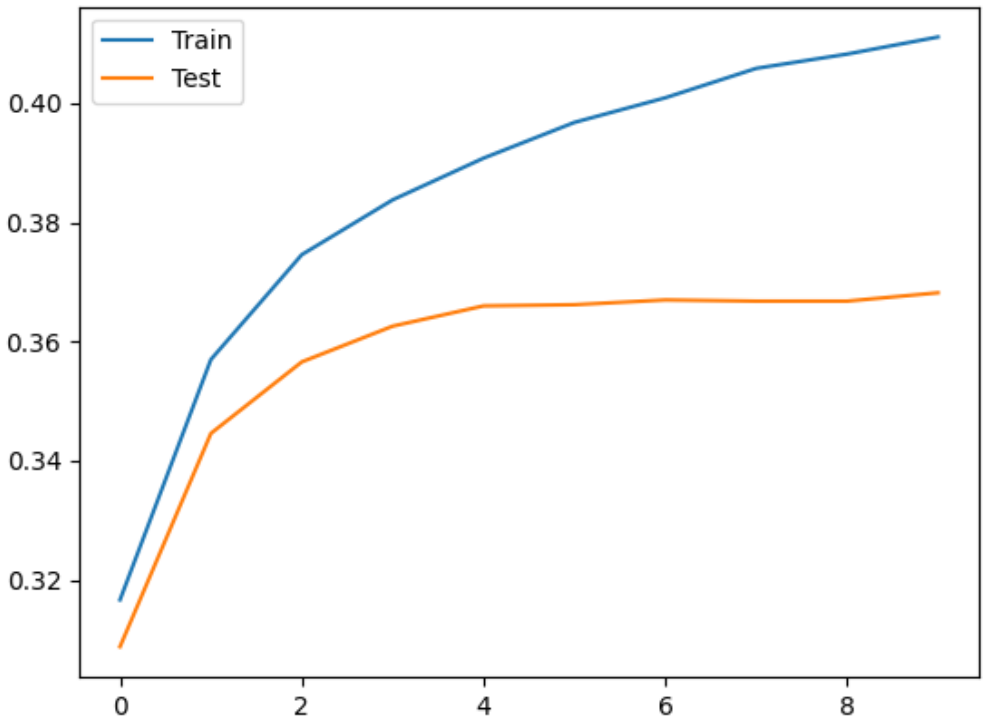


3. Adding Sparsity to Model

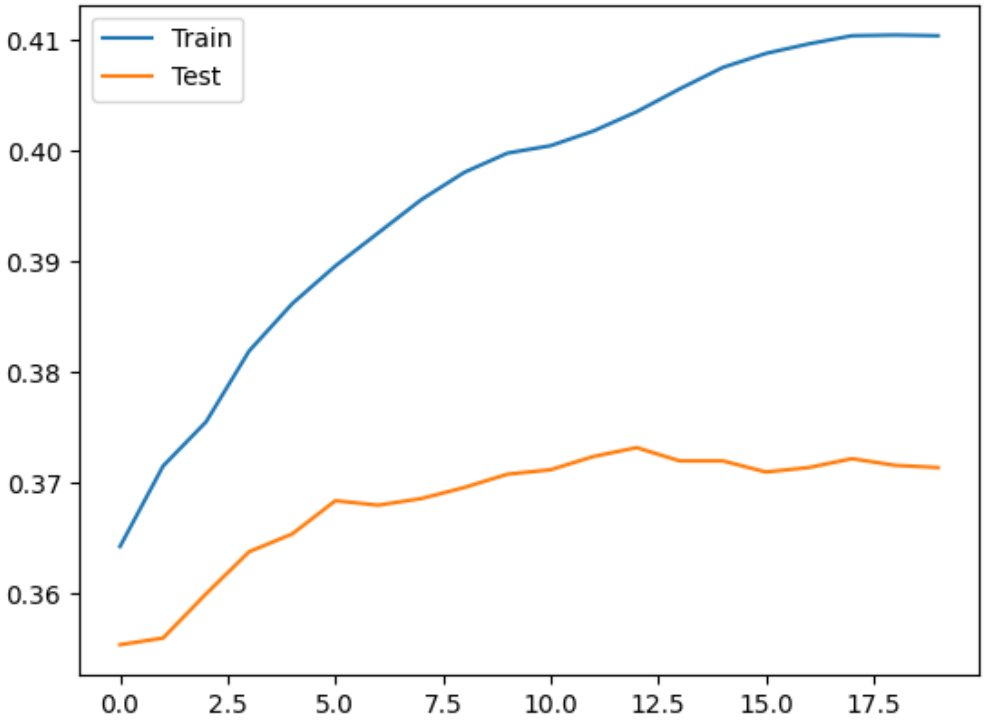
The first thing that we can do is crank up the `l1` regularization and see if we can get a more sparse model. When doing this we can plot the training and test accuracy using the same script as above. When setting `C=0.0005` we get a sparsity of 3.82%, but we take a huge hit to our accuracy, as shown in the plot below.



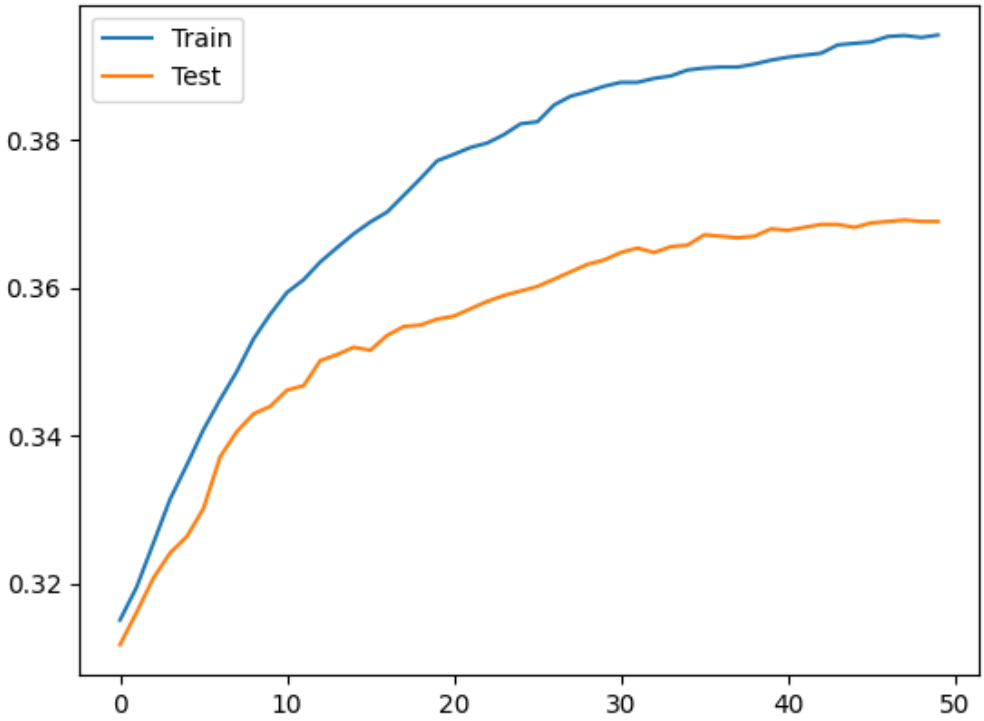
This is a significant tank to our performance. So maybe its not the best to use Lasso to sparsity the data. Instead we can use dropout to create sparsity. Within scikit we can simply just set `coef_` after every epoch to keep dropout throughout training. We will start by dropping out the first 90 columns and last 30 columns. After doing that we get a sparsity of 3.92% and the accuracy is shown below.



This is essential just removing the information from the top and bottom rows of the matrix, however we can eliminate random columns from the matrix and see how the model preforms. When zeroing 60% of the columns we get a total sparsity of 60.55% and a model accuracy is shown below.



Finally we can bump the sparsity up to 85% and here we finally see some losses in the model accuracy, where we get an accuracy of about `.35` which is only `.02` less then our original model. The plot for the training accuracy can be seen below. The plot was generated by the following script.


Train
Test
0.38
0.36
0.34
0.32
0
10
20
30
40
50

```
lg = LogisticRegression(multi_class='multinomial',penalty='l1',C=0.001,solver='saga', verbose=1,
n_jobs=-1,max_iter=1,warm_start=True,random_state=42)
# Calculate the number of columns to eliminate (40% of total columns)
num_columns_to_eliminate = int(0.85 * 3072)

# Generate random column indices to eliminate
random_indices = np.random.choice(3072, size=num_columns_to_eliminate, replace=False)
epochs = 50
training_loss = []
validation_loss = []
for epoch in range(epochs):
    lg = lg.fit(X_train, y_train)
    # lg.coef_[:, :30] = 0
    # lg.coef_[:, -30:] = 0
    # Set the columns specified by random indices to zero
    lg.coef_[:, random_indices] = 0
    Y_pred = lg.predict(X_train)
    curr_train_score = accuracy_score(y_train, Y_pred)   # training performances
    Y_pred = lg.predict(X_test)
    curr_valid_score = accuracy_score(y_test, Y_pred)    # validation performances
    training_loss.append(curr_train_score)               # list of training perf to plot
    validation_loss.append(curr_valid_score)             # list of valid perf to plot


plt.plot(range(epochs),training_loss,label='Train')
plt.plot(range(epochs),validation_loss,label='Test')
plt.legend()
plt.show()

coef_lg = lg.coef_.ravel()
sparsity_lg = np.mean(coef_lg == 0) * 100
print("{:<40} {:.2f}%".format("Sparsity with L1 penalty:", sparsity_lg))
```

Problem 2

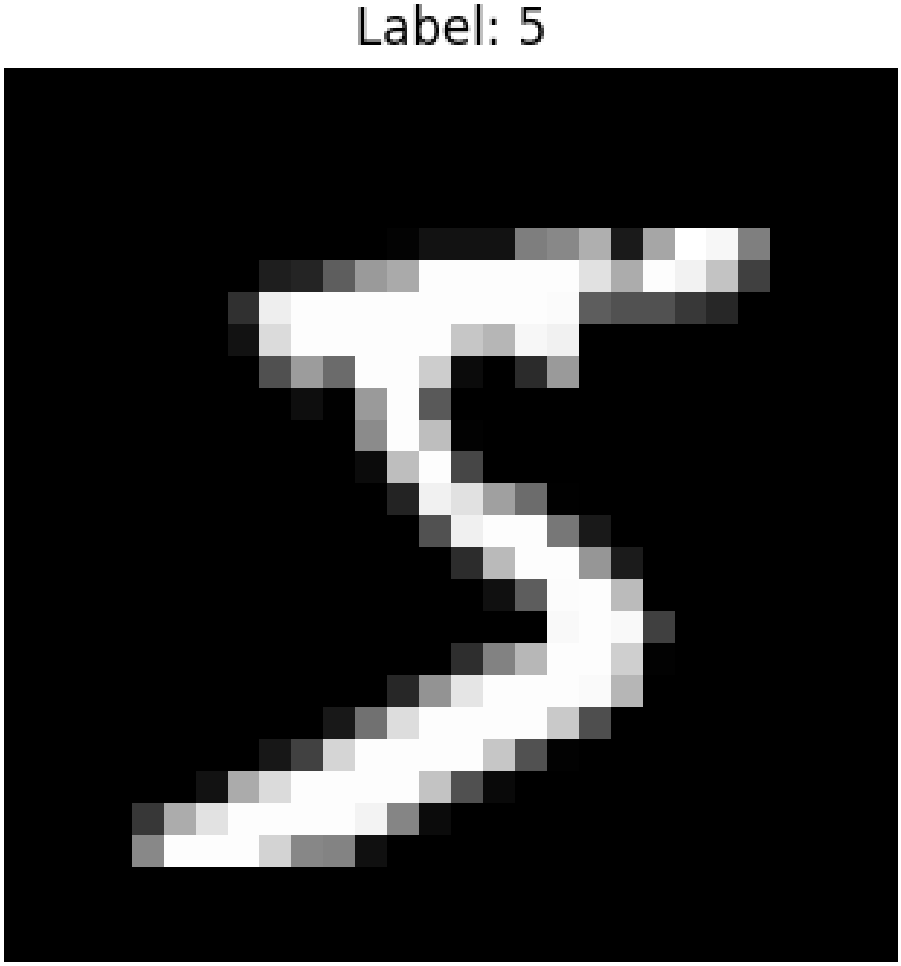
1. Setup

We will start by fetching MNIST using `fetch_openml` from scikitlearn. We can run the following script in order to get the dataset.

```
# Fetch the MNIST dataset
mnist = fetch_openml(name="mnist_784",version=1,parser='auto')

# Access the data and target labels
X, y = mnist.data, mnist.target

print(X.shape,y.shape)
```

We can then do the same thing as last time and display one of the MNIST images using the following script.

```
# Check to make sure that we are getting images
def display_mnist_image(image_data, label):
    # Reshape the image data to 28x28 pixels
    image_data = image_data.reshape(28, 28)

    plt.figure(figsize=(4, 4))  # Set the figure size
    plt.imshow(image_data, cmap='gray')  # Display the image in grayscale
    plt.title(f"Label: {label}")  # Display the label as the title
    plt.axis('off')  # Turn off axis labels and ticks
    plt.show()

display_mnist_image(X[0],y[0])
```



2. Model

We can start off by running a boilerplate model and seeing what results we get. We can then tune the hyperparameters to try to get the best model. We will start off with this as our base model.

```
# Create LogisticRegression Model
lg = LogisticRegressionCV(
    multi_class='multinomial',
    penalty='elasticnet',
    cv=cv_splitter,
    Cs=1,
    solver='saga',
    verbose=1,
    n_jobs=-1,
    l1_ratios=[.5],
    max_iter=40,
    random_state=42
)

# Fit the model to the training data
lg.fit(X_train,y_train)

# Predict probabilities on the test and training data
scores_test = lg.predict_proba(X_test)
scores_train = lg.predict_proba(X_train)

# Calculate the log loss
logloss_test = log_loss(y_test, scores_test)
logloss_train = log_loss(y_train, scores_train)

# Display the log loss for test and training data
print("Log Loss (Test Data):", logloss_test)
print("Log Loss (Training Data):", logloss_train)
```

After running this we get a categorical crossentropy of 0.3323 for the test set and 0.3131 for the training set. However we can tune the model a little bit and we can get a categorical crossentropy of 0.304 for the test data and 0.22104 for the training data. To get these better crossentropy we will have our `LogisticRegressionCV` model search for the best `C` to regularize the data with. Note that this would often take a long time, however `LogisticRegressionCV` automatically parallelizes onto the CPU, which speeds it up so it only takes 9.5 mins.

```
# Create LogisticRegression Model
lg = LogisticRegressionCV(
    multi_class='multinomial',
    penalty='elasticnet',
    cv=cv_splitter,
    Cs=[0.01,0.05,0.1,.5,1,2],
    solver='saga',
    verbose=1,
    n_jobs=-1,
    l1_ratios=[.25,.5,.75],
    max_iter=40,
    random_state=42
)

# Fit the model to the training data
lg.fit(X_train,y_train)

# Predict probabilities on the test and training data
scores_test = lg.predict_proba(X_test)
scores_train = lg.predict_proba(X_train)

# Calculate the log loss
logloss_test = log_loss(y_test, scores_test)
logloss_train = log_loss(y_train, scores_train)

# Display the log loss for test and training data
print("Log Loss (Test Data):", logloss_test)
print("Log Loss (Training Data):", logloss_train)
```

Output

```
Log Loss (Test Data): 0.3041792937608247
Log Loss (Training Data): 0.22104230408308853
Best C: 0.05
Best l1_ratio: 0.75
```

The next thing we can do is try to increase the sparsity of our model. We can start off by increasing the l_1 penalty to enforce sparseness. We can use the following script to do this, and we can run it over several l_1 values.

```
# Define a list of candidate C values to iterate over
C_values = [0.001, 0.01, 0.1, 1.0]

# Iterate over each C value and fit a logistic regression model
for C in C_values:
    # Create a LogisticRegression model with the current C value
    model = LogisticRegression(
        C=C,
        multi_class='multinomial',
        penalty='l1',  # Use 'l1' penalty for logistic regression
        solver='saga',  # Use 'saga' solver for logistic regression
        max_iter=40,   # Specify the maximum number of iterations
        verbose=1,
        n_jobs=-1,
        random_state=42
    )

    # Fit the model to the data
    model.fit(X_train, y_train)

    # Calculate the log loss
    scores = model.predict_proba(X_test)
    log_loss_value = log_loss(y_test, scores)

    # Calculate sparsity as a percentage
    sparsity_percentage = np.mean(model.coef_ == 0) * 100

    # Print the log loss and sparsity for the current C value
    print(f"For C={C:.4f} - Log Loss: {log_loss_value:.4f}, Sparsity: {sparsity_percentage:.2f}%")
```

Output

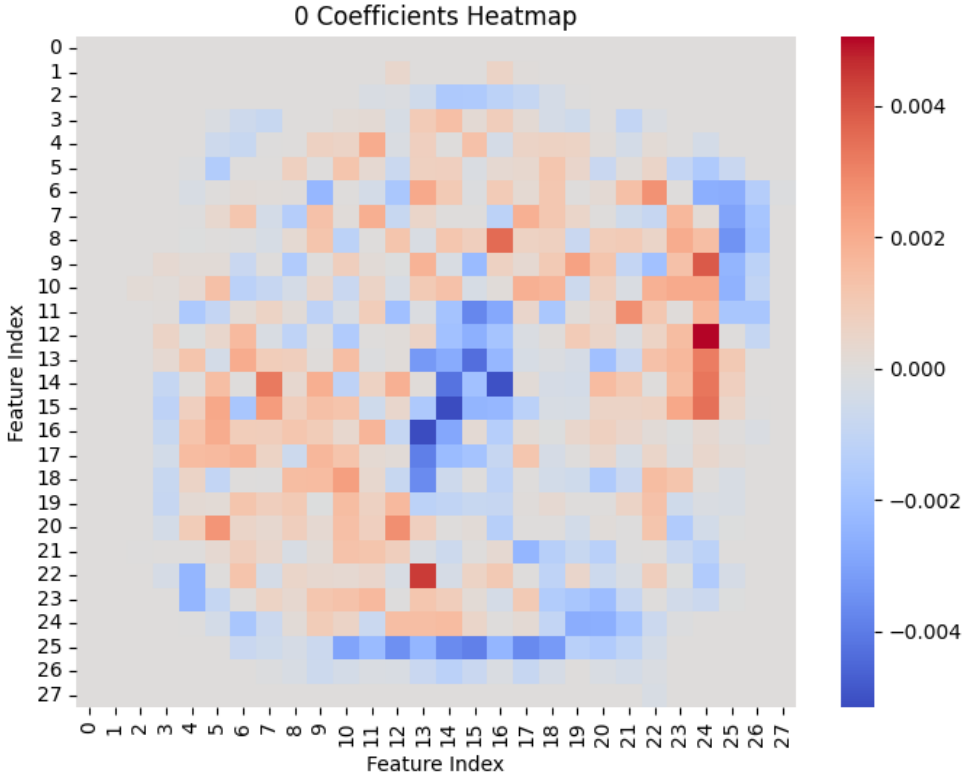
```
For C=0.0001 - Log Loss: 0.3732, Sparsity: 86.17%
For C=0.0010 - Log Loss: 0.2925, Sparsity: 65.11%
For C=0.0100 - Log Loss: 0.2947, Sparsity: 31.73%
For C=0.1000 - Log Loss: 0.2980, Sparsity: 21.59%
For C=1.0000 - Log Loss: 0.2984, Sparsity: 16.59%
```

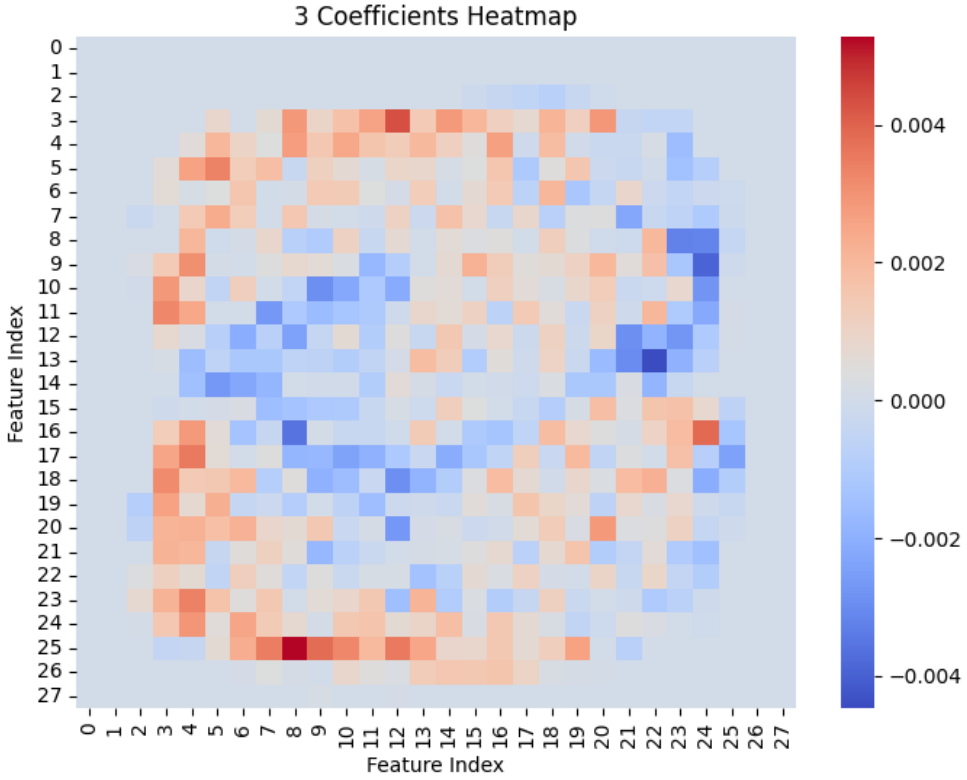
The funny thing is that we get better categorical crossentropy using only `l1` regularization rather than `elasticnet` regularization. We can get a really good categorical crossentropy loss even when having a high Sparsity, which makes sense as only a few pixels are really needed to identify the image.

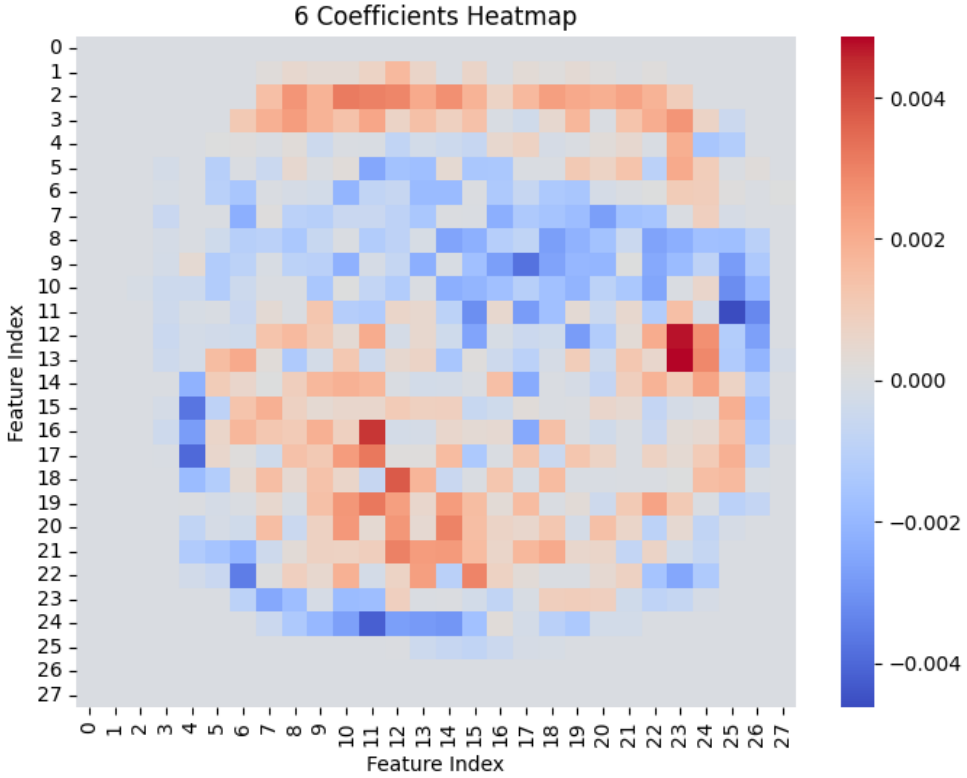
Now that we have a sparser solutions we can print the weights of our `LogisticRegression` model in order to see if we can visualize what it is doing. We can use a similar script as displaying our images from the dataset.

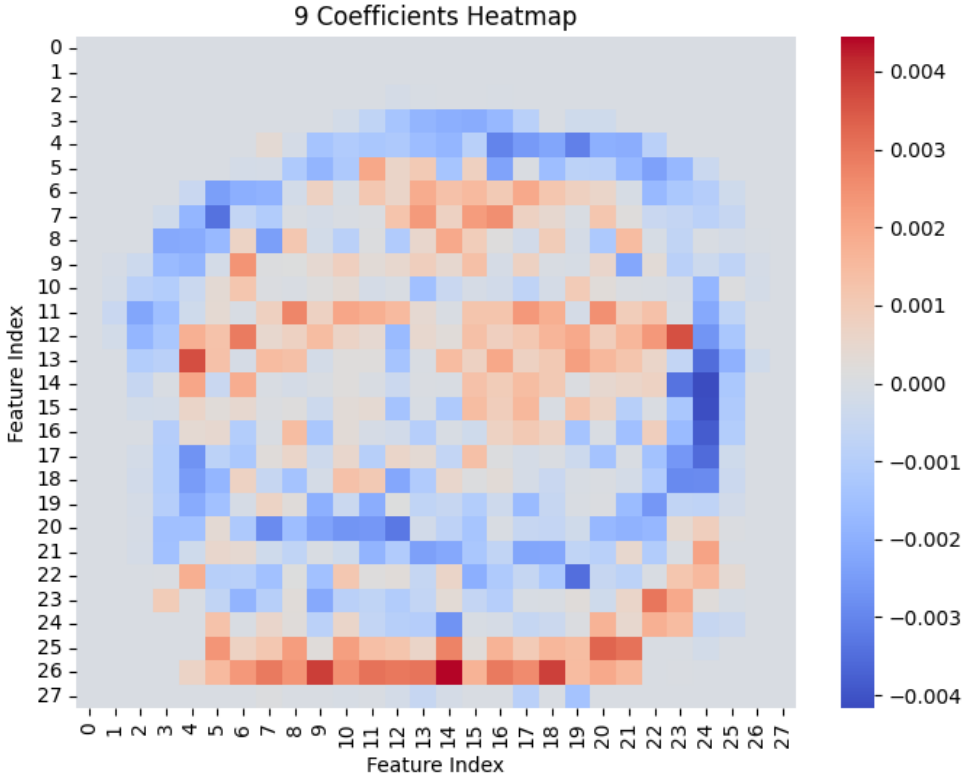
```
def plot_coefficients_heatmap(coefficients,classifier_name):
    # Reshape the coefficients into a 28x28 grid
    coefficients = coefficients.reshape(28, 28)

    # Create a heatmap using Seaborn
    plt.figure(figsize=(8, 6))  # Set the figure size
    sns.heatmap(coefficients, annot=False, cmap='coolwarm', cbar=True)

    # Add labels and title
    plt.xlabel("Feature Index")
    plt.ylabel("Feature Index")
    plt.title(f"{classifier_name} Coefficients Heatmap")

    plt.show()

for val in range(10):
    plot_coefficients_heatmap(model.coef_[val],str(val))
```

For the C value of 0.01, we get the following heatmaps for each of the categories. I picked out some of the most notable heatmaps.


0 Coefficients Heatmap
Feature Index
Feature Index
0.004
0.002
0.000
−0.002
−0.004


3 Coefficients Heatmap
Feature Index
Feature Index
0.004
0.002
0.000
−0.002
−0.004





One final thing that we should do is to check the accuracy of our model. We can do that with the following script.

```
# Getting the accuracy of the model
model = LogisticRegression(
    C=0.001,
    multi_class='multinomial',
    penalty='l1',  # Use 'l1' penalty for logistic regression
    solver='saga',  # Use 'saga' solver for logistic regression
    max_iter=40,   # Specify the maximum number of iterations
    verbose=1,
    n_jobs=-1,
    random_state=42
)

# Perform 5-fold cross-validation and obtain predicted probabilities
score = cross_val_score(model, X, y, cv=5)

# Calculate and print the mean log loss
print(f"The Accuracy: {score.mean():.4f}")
```

Output

```
The Accuracy: 0.9204
```

Problem 3

We will now try to use `RandomForests` and `GradientBoosting` to try to get a better model on MNIST. We will start by setting up the environment and then we can tune the hyperparameters of the models.

1. Setup

We will just copy the script from above to set up the environment.

```
# Fetch the MNIST dataset
mnist = fetch_openml(name="mnist_784",version=1,parser='auto')

# Access the data and target labels
X, y = mnist.data.to_numpy(), mnist.target.to_numpy()

print(X.shape,y.shape)

# Create split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.25, random_state=42)
```

2. Random Forest

We first begin by creating a boilerplate `RandomForest` and see what kind of accuracy we can get. Our initial `RandomForest` can be seen by the code below.

```
# Create a Random Forest classifier
rf_classifier = RandomForestClassifier(
    n_estimators=100,
    max_depth=8,
    min_samples_split=4,
    min_samples_leaf=2,
    n_jobs=-1,
    verbose=1,
    random_state=42
)

# Train the classifier on the training data
rf_classifier.fit(X_train, y_train)

# Make predictions on the test data
y_pred = rf_classifier.predict(X_test)

# Calculate accuracy
accuracy = accuracy_score(y_test, y_pred) * 100
print("Accuracy:", accuracy)
```

From this we get a score of 92.4%, which is already so much better than our `LogisticRegression` model. We can tune the hyperparameters to see if we can get a better score. We ended up with the following model below which had an accuracy of 96.5%.

```
# Create a Random Forest classifier
rf_classifier = RandomForestClassifier(
    n_estimators=170,
    max_depth=25,
    min_samples_split=4,
    min_samples_leaf=2,
    n_jobs=-1,
    verbose=1,
    random_state=42
)

# Train the classifier on the training data
rf_classifier.fit(X_train, y_train)

# Make predictions on the test data
y_pred = rf_classifier.predict(X_test)

# Calculate accuracy
accuracy = accuracy_score(y_test, y_pred) * 100
print("Accuracy:", accuracy)
```

However we should use `cross_validation` in order to make sure that they results are reasonable. We can do that by running the following script.

```
# Perform 5-fold cross-validation and obtain accuracy scores
cv_scores = cross_val_score(rf_classifier, X, y, cv=5)

# Print the cross-validated accuracy scores
for fold, accuracy in enumerate(cv_scores, start=1):
    print(f"Fold {fold} Accuracy: {accuracy:.4f}")

# Calculate and print the mean accuracy
mean_accuracy = cv_scores.mean()
print(f"Mean Accuracy: {mean_accuracy:.4f}")
```

Output

```
Fold 1 Accuracy: 0.9665
Fold 2 Accuracy: 0.9656
Fold 3 Accuracy: 0.9659
Fold 4 Accuracy: 0.9635
Fold 5 Accuracy: 0.9707
Mean Accuracy: 0.9665
```

We can also change this script to show the loss.

```
# Perform 5-fold cross-validation and obtain predicted probabilities
y_pred_proba = cross_val_predict(rf_classifier, X, y, cv=5, method='predict_proba')

# Calculate the log loss
logloss = log_loss(y, y_pred_proba)

# Calculate and print the mean log loss
print(f"The Log Loss: {logloss:.4f}")
```

Output

```
The Log Loss: 0.2706
```

As you can see `RandomForest` does a lot better than the `LogisticRegression` when comparing the categorical crossentropy loss. This is also evident when looking at the accuracy of each of these models with `RandomForest` having an accuracy of 96.6% while `LogisticRegression` has an accuracy of 92.1%.

3. Gradient Boosting

Next we can use `GradientBoosting` on the MNIST dataset and see if we can get a better model than `RandomForest` or `LogisticRegression`. We will use the following initial model to test accuracy, which we can then explore further from.

```
# Create a XGBoost Classifier
xgb_classifier = xgb.XGBClassifier(
    n_estimators=170,      # Number of boosting rounds (trees)
    learning_rate=0.1,     # Step size shrinking to prevent overfitting
    max_depth=8,           # Maximum depth of individual trees
    n_jobs=-1,
    verbosity=2,
    random_state=42        # Random seed for reproducibility
)

# Train the classifier on the training data
xgb_classifier.fit(X_train, y_train)

# Make predictions on the test data
y_pred = xgb_classifier.predict(X_test)

# Calculate accuracy
accuracy = accuracy_score(y_test, y_pred) * 100
print("Accuracy:", accuracy)
```

Output

```
Accuracy: 97.44571428571429
```

That's a pretty good starting score, but we will go ahead and try to get a better score with hyperparameter tuning. We will go through an adjust `max_depth`, `learning_rate`, and `n_estimators` to try to get a better model. We will also incorporate early stopping in order to try to prevent overfitting. After some model tuning we landed on the following hyperparameters, displayed in the script below.

```
# Create a XGBoost Classifier
xgb_classifier = xgb.XGBClassifier(
    n_estimators=700,      # Number of boosting rounds (trees)
    learning_rate=1,       # Step size shrinking to prevent overfitting
    max_depth=6,           # Maximum depth of individual trees
    n_jobs=-1,
    verbosity=2,
    random_state=42        # Random seed for reproducibility
)

# Train the classifier on the training data
xgb_classifier.fit(X_train, y_train)

# Make predictions on the test data
y_pred = xgb_classifier.predict(X_test)

# Calculate accuracy
accuracy = accuracy_score(y_test, y_pred) * 100
print("Accuracy:", accuracy)
```

Output

```
Accuracy: 97.89714285714285
```

We will also use 5-fold cross validation to ensure that these results are reasonable. To do that, we can run the following script.

```
# Perform 5-fold cross-validation and obtain accuracy scores
cv_scores = cross_val_score(rf_classifier, X, y, cv=5)

# Print the cross-validated accuracy scores
for fold, accuracy in enumerate(cv_scores, start=1):
    print(f'Fold {fold} Accuracy: {accuracy:.4f}')

# Calculate and print the mean accuracy
mean_accuracy = cv_scores.mean()
print(f'Mean Accuracy: {mean_accuracy:.4f}')
```

Output

```
Mean Accuracy: 97.9102
```

We can also do this to show the loss with the following script and we get.

```
# Perform 5-fold cross-validation and obtain predicted probabilities
y_pred_proba = cross_val_predict(rf_classifier, X, y, cv=5, method='predict_proba')

# Calculate the log loss
logloss = log_loss(y, y_pred_proba)

# Calculate and print the mean log loss
print(f"The Log Loss: {logloss:.4f}")
```

Output

```
The Log Loss: 0.0728
```

As we can see `XGBoost` had the best model compared to `RandomForest` and `LogisticRegression` for the MNIST dataset. However we will eventually see that CNNs will preform a lot better than all of these. Note for the future that this took forever to run using `XGBoost` and other boosting methods should be used.

Problem 4

We will now try to use `RandomForests` and `GradientBoosting` to try to get a better model on CIFAR-10. We will start by setting up the environment and then we can tune the hyperparameters of the models.

1. Setup

We will just copy the script from above to set up the environment.

```
# Fetch the CIFAR-10-Small dataset
cifar_10_small = fetch_openml(name="CIFAR_10_small", version=1,parser='auto')

# Access the data and target labels
X, y = cifar_10_small.data, cifar_10_small.target

print(X.shape,y.shape)

# Create split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.25, random_state=42)
```

2. Random Forest

We can start by using the same `RandomForest` of the MNIST set as a baseline to begin exploring for a better model for CIFAR-10. We can run the following script and see what kind of accuracy we get.

```
# Create a Random Forest classifier
rf_classifier = RandomForestClassifier(
    n_estimators=170,
    max_depth=25,
    min_samples_split=4,
    min_samples_leaf=2,
    n_jobs=-1,
    verbose=1,
    random_state=42
)

# Train the classifier on the training data
rf_classifier.fit(X_train, y_train)

# Make predictions on the test data
y_pred = rf_classifier.predict(X_test)

# Calculate accuracy
accuracy = accuracy_score(y_test, y_pred) * 100
print("Accuracy:", accuracy)
```

Output

```
Accuracy: 44.56
```

Better than our `LogisticRegression` model, but still not that great. We will do a decent amount of hyperparameter tuning in order to come up with the best model. For tuning this mode I tried different depths and different number of estimators but found that a depth of 25 was pretty good, but the number of estimators need to be larger. After some hyperparameter tuning we get the following model, which is displayed in the script below.

```
# Create a Random Forest classifier
rf_classifier = RandomForestClassifier(
    n_estimators=400,
    max_depth=25,
    min_samples_split=4,
    min_samples_leaf=2,
    n_jobs=-1,
    verbose=1,
    random_state=42
)
```

Output

```
Accuracy: 45.28
```

We will again use 5-fold cross validation in order to ensure that out model accuracy is reasonable. We can use the same scripts in the MNIST sections and we get the following outputs.

Accuracy Output

```
Fold 1: Accuracy = 0.4562
Fold 2: Accuracy = 0.4462
Fold 3: Accuracy = 0.4440
Fold 4: Accuracy = 0.4590
Fold 5: Accuracy = 0.4490
Mean Accuracy = 0.4509
```

Loss Output

```
The Log Loss: 1.6851
```

3. Gradient Boosting

We can again start by using the `XGBoost` model from the MNIST dataset to determine a baseline accuracy and then tune the model from their. However one thing that we will do in order to reduce the computational complexity of our models is to use `LightGBM` instead of `XGBoost`. The script for this model can be seen below.

```
# Create a LightGBM classifier
lgb_classifier = lgb.LGBMClassifier(
    n_estimators=600,      # Number of boosting rounds (trees)
    learning_rate=0.1,     # Step size shrinking to prevent overfitting
    max_depth=7,           # Maximum depth of individual trees
    n_jobs=-1,             # Use all available CPU cores
    verbosity=1,           # Verbosity level
    random_state=42        # Random seed for reproducibility
)

# Train the classifier on the training data
lgb_classifier.fit(X_train, y_train)

# Make predictions on the test data
y_pred = lgb_classifier.predict(X_test)

# Calculate accuracy
accuracy = accuracy_score(y_test, y_pred) * 100
print("Accuracy:", accuracy)
```

Output

```
Accuracy: 53.82
```

We will do some hyperparameter tuning to try to improve the model score, however due to the high computation complexity of using gradient boosting, we will not be able to an exhaustive search to find the bets hyperparameters. After tuning the hyperparameters we got the following model.

```
# Create a LightGBM classifier
lgb_classifier = lgb.LGBMClassifier(
    n_estimators=650,      # Number of boosting rounds (trees)
    learning_rate=0.1,     # Step size shrinking to prevent overfitting
    max_depth=6,           # Maximum depth of individual trees
    n_jobs=-1,             # Use all available CPU cores
    verbosity=1,           # Verbosity level
    random_state=42        # Random seed for reproducibility
)
```

Output

```
Accuracy: 54.379999999999995
```

While I would normally run scripts to show cross-validation and ensure the results of our model. Due to the shear amount of processing power required in order to do this, we passed on this one.

However, again we can see that `XGBoost` did the best compared to `RandomForest` and `LogisticRegression`.

Problem 5

1. PyTorch Logistic Regression

The tutorial given lead us through creating better models for MNIST using pytorch starting from a logistic regression, using softmax.

```
weights = torch.randn(784, 10) / math.sqrt(784)
weights.requires_grad_()
bias = torch.zeros(10, requires_grad=True)

def log_softmax(x):
return x - x.exp().sum(-1).log().unsqueeze(-1)

def model(xb):
return log_softmax(xb @ weights + bias)
```

We then quickly proceed to a CNN.

```
class Mnist_CNN(nn.Module):
def __init__(self):
super().__init__()
self.conv1 = nn.Conv2d(1, 16, kernel_size=3, stride=2, padding=1)
self.conv2 = nn.Conv2d(16, 16, kernel_size=3, stride=2, padding=1)
self.conv3 = nn.Conv2d(16, 10, kernel_size=3, stride=2, padding=1)


def forward(self, xb):
xb = xb.view(-1, 1, 28, 28)
xb = F.relu(self.conv1(xb))
xb = F.relu(self.conv2(xb))
xb = F.relu(self.conv3(xb))
xb = F.avg_pool2d(xb, 4)
return xb.view(-1, xb.size(1))
```

---

2. PyTorch CNN

The best CNN (full code in folder) we came up with built off the original architecture but adjective epoch, batch size, kernel_size, and more to get an average validation loss of 0.086 and accuracy of .987

```
model = nn.Sequential(
#major accuracy jump after changing kernel_size
nn.Conv2d(1, 16, kernel_size=5, stride=2, padding=1),
nn.ReLU(),
nn.Conv2d(16, 16, kernel_size=5, stride=2, padding=1),
nn.ReLU(),
nn.Conv2d(16, 10, kernel_size=5, stride=2, padding=1),
nn.ReLU(),
nn.AdaptiveAvgPool2d(1),
Lambda(lambda x: x.view(x.size(0), -1)),
)
```

There was an especially large jump in accuracy after kernel_size was adjusted as seen from output before:

Output Before

```
Epoch: 0 Val loss: 0.30684129095077517 Accuracy: tensor(0.8750)
Epoch: 1 Val loss: 0.26993565064668656 Accuracy: tensor(0.9375)
Epoch: 2 Val loss: 0.20180223822891713 Accuracy: tensor(1.)
Epoch: 3 Val loss: 0.19898630271553994 Accuracy: tensor(0.8750)
Epoch: 4 Val loss: 0.16966740633025765 Accuracy: tensor(0.9375)
```

Output After

```
Epoch: 0 Val loss: 0.11342142640054226 Accuracy: tensor(1.)
Epoch: 1 Val loss: 0.08887874217927456 Accuracy: tensor(1.)
Epoch: 2 Val loss: 0.08536216458976269 Accuracy: tensor(1.)
Epoch: 3 Val loss: 0.07486423498317599 Accuracy: tensor(0.9375)
Epoch: 4 Val loss: 0.06626780612738803 Accuracy: tensor(1.)
```