Carnegie Mellon Unversity
17-644: Applied Deep Learning
Assignment# 3 Report

A. Dataset Description

The dataset I selected for this assignment is MNIST Handwritten Digits, as suggested in the previous assignment. I didn't want to switch my final project dataset since I thought the fact that MNIST Handwritten Digits is a structured dataset could help my learning. Also, my targeted dataset for the final project isn't ready yet. I have some ideas that I'll be sharing in the last part of this report. MNIST dataset contains 60,000 training and 10,000 testing grayscale images of handwritten digits (0–9). Each image is 28×28 pixels (784 total features when flattened 1D), with pixel values ranging from 0 to 255. I will normalize the images to the 0–1 range before feeding them to the model.

B. Model Summary & Performance

Architecture:

- **Input**: Flattened 28×28 image = 784 neurons (in 1D vector format)
- **Hidden Layer 1:** Dense (128 units, ReLU)
- **Hidden Layer 2:** Dense (64 units, ReLU)
- **Output Layer:** Dense (10 units, Softmax)

# of Learnable Parameters: 109,386

- Calculated manually and verified via `model.summary()`

$$Q = \sum_{i=1}^{2} n_i(n_{i-1} + 1)$$

For the first hidden layer;

$$Q1 = (784\ x\ 128) + 128 = 100{,}480$$

For the second hidden layer;

$$Q2 = (128\ x\ 64) + 64 = 8{,}256$$

For the output layer;

$$Q3 = (64\ x\ 10) + 10 = 650$$

In total;

$$Q = Q1 + Q2 + Q3$$
$$\boldsymbol{Q = 109{,}386}$$

```
Layer (type)                 Output Shape              Param #
=================================================================
 flatten (Flatten)           (None, 784)               0

 dense (Dense)               (None, 128)               100480

 dense_1 (Dense)             (None, 64)                8256

 dense_2 (Dense)             (None, 10)                650

=================================================================
Total params: 109,386
Trainable params: 109,386
Non-trainable params: 0
_________________________________________________________________
```

Code 1 Neural network model initialization code and output log

Hyperparameters:

- **Loss**: sparse_categorical_crossentropy
- **Optimizer**: ADAM
- **Batch size**: 64
- **Epochs**: 20 (But then early stopped at 3 to prevent overfitting)

Performance (Figure 1):

- **Training Accuracy**: appx. 99.7%
- **Validation Accuracy**: appx. 97.7%
- **Evidence of Overfitting**: Training loss approaches zero while validation loss starts rising after about 3 epochs. The widening gap suggests overfitting. That's why, I suggested an early stopping at 3 epochs.

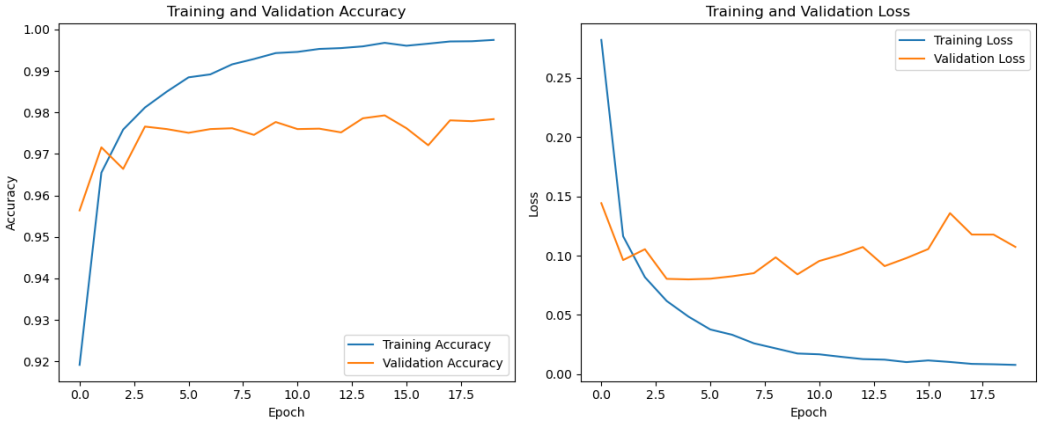

Figure 1 Training and validation accuracy and validation loss plot

C. Model Summary & Performance (Unregularized)

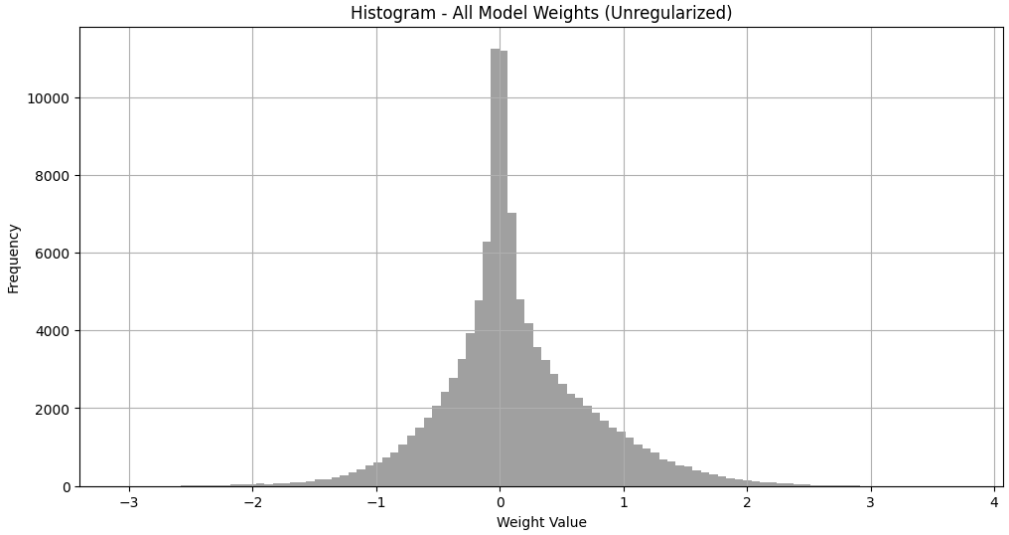

Figure 2 Histogram of weights for all dense layers, with attributes, unregularized

```
all_weights = []
for layer in model.layers:
    if hasattr(layer, 'kernel'):  # check if layer has weights
        weights = layer.get_weights()[0].flatten() # flatten to an 1d array
        all_weights.extend(weights)
```

```
plt.figure(figsize=(12, 6))
plt.hist(all_weights, bins=100, color='gray', alpha=0.75)
plt.title("Histogram – All Model Weights (Unregularized)")
plt.xlabel("Weight Value")
plt.ylabel("Frequency")
plt.grid(True)
plt.show()
```

Code 2 Layer weights histogram plot

The bell-shaped curve is expected since I didn't use any regularization algorithm to prevent overfitting. The symmetric bell shape, which starts at zero, suggests the ReLU network algorithm started weights small and incremented a little only. I assume this is because of the lack of regularization. As we learned in the lecture, regularization is useful to develop generalizable models since it constrains weights, leading to a narrower distribution and larger generalization.

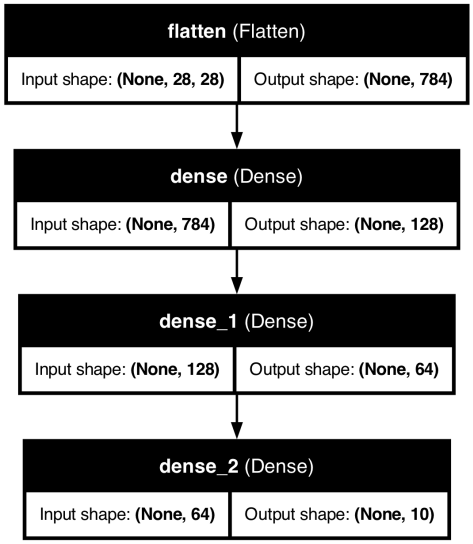

Figure 3 plot_model output to understand my neural network architecture to help me visualize it before starting to plotting each layer's weights

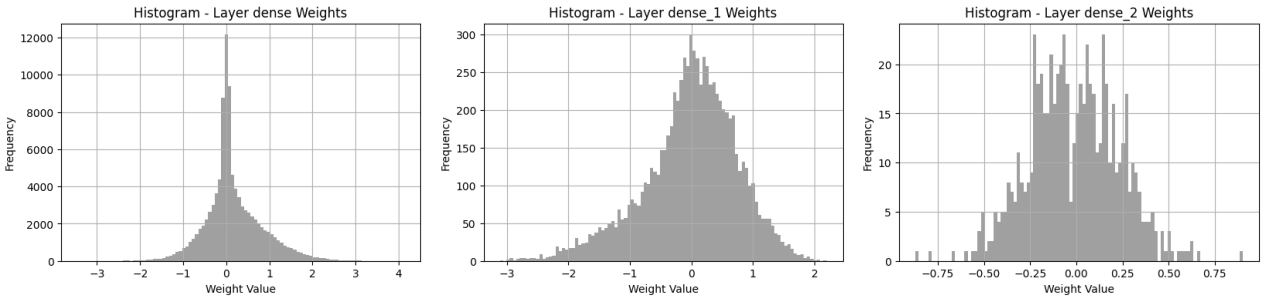

Figure 4 Individual layers' weight histograms to better understand what's going on in each layer

The distribution across each layer changes significantly, as depicted in Figure 4. I began by examining my model architecture (Figure 3), which consists of three dense layers. To better understand each layer's role, I first visualized their individual weight histograms. The three visualizations clearly emphasize the discrepancy between layers. The first dense layer (Layer 2) has the widest distribution, spanning from approximately -3 to +4. This is expected, as it directly connects to the 784 raw pixel inputs, bearing most of the representational load in early learning. The second dense layer (Layer 3) displays a more concentrated weight

range, roughly between -3 and +2, indicating that it builds on extracted features rather than raw pixel information. Finally, the output layer (Layer 4) has an extremely narrow distribution, clustered tightly around zero. This is logical—since it maps 64 neurons to just 10 class scores, the required transformations are small and task-specific. Overall, the variation across layers aligns with each layer's purpose in the model pipeline.

D. Regularization & Re-training

I decided to apply L2 regularization, which penalizes large weights by adding a cost term, as described in the lecture notes:

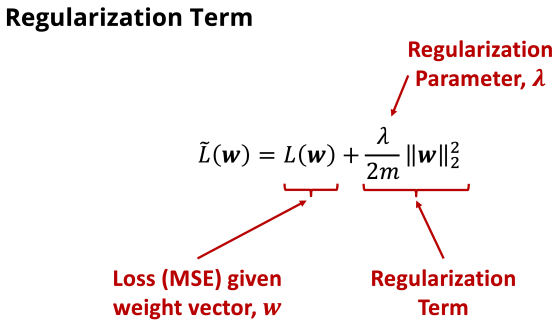

Figure 6 L2 regularization regularization term, from Applied Deep Learning lecture notes, CMU, Prof. Clarence Worrell

This regularization logic will encourage the model to find simpler and smoother solutions, which is expected to lead to tighter and more compact weight distributions. I also expect to increase accuracy and decrease overfitting.

```
from tensorflow.keras import regularizers

# same architecture with L2 regularization
model_l2 = keras.Sequential([
    layers.Flatten(input_shape=(28, 28)),
    layers.Dense(128, activation='relu', kernel_regularizer=regularizers.l2(0.01)),
    layers.Dense(64, activation='relu', kernel_regularizer=regularizers.l2(0.01)),
    layers.Dense(10, activation='softmax')
])
model_l2.compile(optimizer='adam',
                 loss='sparse_categorical_crossentropy',
                 metrics=['accuracy'])

# train the regularized model
history_l2 = model_l2.fit(X_train_normalized, y_train,
                          epochs=20,
                          validation_data=(X_test_normalized, y_test),
                          batch_size=64)
```

Code 3 Training the model with L2 regularization added

E. Model Summary & Performance (L2 Regularized)

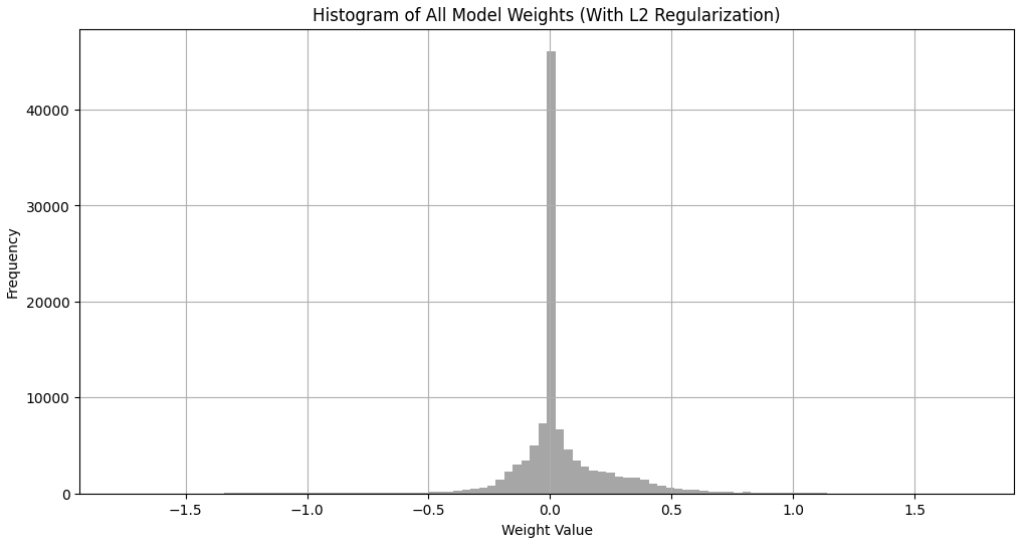

Figure 6 Histogram of weights for all dense layers, with attributes, after L2 regularization

Comparing the regularized mode weight histogram with Figure 2, unregularized weight histogram, I observed a sharper peak at 0 (zero) since regularization seems to squeeze the weights toward zero. I assume regularization reduced model complexity and helped generalization. Unlike the unregularized model which had weights ranging up to -3 to 3, this one is mostly within -0.5 to 0.5, with very few outliers. That's what L2 penalty encourages. In addition, The histogram looks much more compact. I expect the smaller, more uniform weights to lead to less overfitting, which is the main goal of L2.

F. Hyperparameter Tuning – Grid Search

For the hyperparameter tuning step, I decided to go with partial grid search since my dataset isn't huge and grid search would allow me to try out many options sequentially and see how better I can carry the performance of my model. I selected three of the search parameters since looking at the whole 9 variations would be computationally and time-wise too expensive.

```
def build_and_train(units_1, units_2, name):
    print(f"\n Training: {name}")
    model = keras.Sequential([
        layers.Flatten(input_shape=(28, 28)),
        layers.Dense(units_1, activation='relu'),
        layers.Dense(units_2, activation='relu'),
        layers.Dense(10, activation='softmax')
    ])
    model.compile(optimizer='adam',
                  loss='sparse_categorical_crossentropy',
                  metrics=['accuracy'])
    history = model.fit(X_train_normalized, y_train,
                        epochs=10,
                        validation_data=(X_test_normalized, y_test),
                        verbose=0)
    val_acc = history.history['val_accuracy'][-1]
    print(f"Final Val Accuracy: {val_acc:.4f}")
    return history, val_acc

# run 3 models
h1, acc1 = build_and_train(64, 32, "Small (64-32)")
h2, acc2 = build_and_train(128, 64, "Original (128-64)")
h3, acc3 = build_and_train(256, 128, "Large (256-128)")

# compare
print("\n Accuracy Comparison:")
print(f"Small  (64-32):   {acc1:.4f}")
print(f"Medium (128-64):  {acc2:.4f}")
print(f"Large  (256-128): {acc3:.4f}")
```

Code 4 Hyperparameter tuning via grid search

I tried three different architectures with grid search-like logic, changing hidden layer sizes in the model:

Model	Hidden Layer Sizes	Notes
model_1	64-32	Smaller baseline
model_2	128-64	Your original model
model_3	256-128	Larger model

Training results were as such:

```
Accuracy Comparison:
Small  (64-32):   0.8477
Medium (128-64):  0.8138
Large  (256-128): 0.8496
```

I trained three variants of the model with different layer sizes: Small (64-32), Original (128-64), and Large (256-128). I know that to conduct a real grid search, I had to try all nine variations, however, due to computational limitations, I decided to select 3 on the graph's diagonal. Surprisingly, the small model achieved nearly the same validation accuracy (84.8%) as the large one (84.9%), suggesting that the task does not require high capacity. The original model performed worse (81.3%), likely due to the suboptimal training dynamics. These imply that tuning hidden layer sizes can result in significant performance changes, and smaller architectures may generalize just as well while being more efficient.

G. Hyperparameter Tuning – Random Search

Since random search is accepted to be more effective than grid search (Figure 7), I decided to try random search as well. I decided to randomize:

- Neurons in layer 1 and layer 2
- Using 3 trials (Since I also conducted 3 in partial grid search due to computational power issues)

CHAPTER 11. PRACTICAL METHODOLOGY

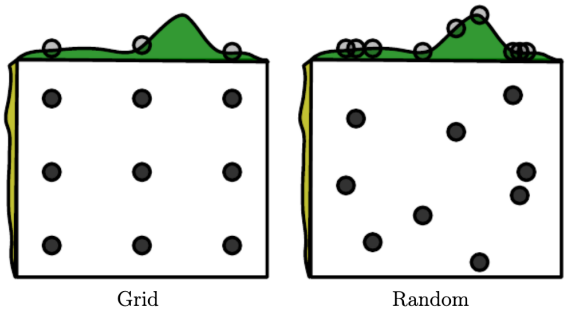

Figure 11.2: Comparison of grid search and random search. For illustration purposes, we display two hyperparameters, but we are typically interested in having many more. *(Left)* To perform grid search, we provide a set of values for each hyperparameter. The search algorithm runs training for every joint hyperparameter setting in the cross product of these sets. *(Right)* To perform random search, we provide a probability distribution over joint hyperparameter configurations. Usually most of these hyperparameters are independent from each other. Common choices for the distribution over a single hyperparameter include uniform and log-uniform (to sample from a log-uniform distribution, take the exp of a sample from a uniform distribution). The search algorithm then randomly samples joint hyperparameter configurations and runs training with each of them. Both grid search and random search evaluate the validation set error and return the best configuration. The figure illustrates the typical case where only some hyperparameters have a significant influence on the result. In this illustration, only the hyperparameter on the horizontal axis has a significant effect. Grid search wastes an amount of computation that is exponential in the number of noninfluential hyperparameters, while random search tests a unique value of every influential hyperparameter on nearly every trial. Figure reproduced with permission from Bergstra and Bengio (2012).

Figure 7 *Grid search vs. Random search.* **Reference**: Goodfellow, I., Bengio, Y., & Courville, A. (2016). Deep learning. MIT Press, p. 428

```
import random

def random_architecture_search(trials=3):
    results = []
    for i in range(trials):
        u1 = random.randint(0, 300)
        u2 = random.randint(0, 150)
        name = f"Trial {i+1} ({u1}-{u2})"
        print(f"\n {name}")
        history, acc = build_and_train(u1, u2, name)
        results.append((name, acc))

    results.sort(key=lambda x: x[1], reverse=True)
    print("\n Best Architectures by Val Accuracy:")
    for name, acc in results:
        print(f"{name}: {acc:.4f}")

random_architecture_search(trials=5)
```

Code 5 Hyperparameter tuning via random search

Training results were as such:

```
Best Architectures by Val Accuracy:
Trial 3 (153-87): 0.8585
Trial 2 (109-30): 0.8545
Trial 1 (104-36): 0.8514
```

Considering my un-tuned model achieved around 81% validation accuracy, the random search yielded a notable improvement. Due to computational constraints, I conducted a partial random search with three trials, testing architectures with varying hidden

layer sizes. The best-performing configuration was Trial 3 (153–87), reaching a validation accuracy of 85.8%, slightly outperforming Trial 2 (109–30) and Trial 1 (104–36). These results suggest that moderately sized and slightly deeper networks may give a better balance between learning capacity and generalization. The improvement over the baseline demonstrates that tuning architecture even with a limited random search can yield meaningful performance gains without resorting to exhaustive grid search.

H. Semester Project Dataset & Description

Research Question:

Can a deep learning model accurately classify building exterior material (e.g., brick, stucco, concrete, frame) from a single street-level image?

Dataset Selection & Motivation:

Initially, I was interested in using the **Bounding Boxes for Urban Perception** dataset introduced in the paper *"Bounding Boxes Are All We Need: Street View Image Classification via Context Encoding of Detected Buildings"* (arXiv:2010.01305). However, the dataset is only hosted on Baidu Drive, a file-hosting service based in China. Since downloading from Baidu requires installing proprietary Chinese software —which some Reddit users even caution against for potential piracy or security concerns— I opted not to proceed with it.

In parallel, I researched several other promising datasets:

- **Building Material Classification (GitHub)** – A well-referenced academic project, but the author is currently on parental leave, and the full dataset isn't publicly accessible. I've reached out via email.
- **Building Façade Typology Dataset (Figshare)** – From a peer-reviewed paper, but limited in scope for my task. (Paper)
- **Pix2Pix Facade Dataset (GitHub)** – Designed for image translation tasks, but lacks metadata. (Project Website)
- **Kaggle Façade Dataset**
- **Roboflow Facades Dataset** – Encountered download issues.
- **CSIRO Dataset** – Found a .txt link list (Data), but downloading is complex and likely incomplete. (Paper)

Updated Plan: Self-Labeled Dataset from RA Project:

Given these challenges, I plan to use a custom-labeled dataset that I've worked on as part of my current research assistantship. This dataset contains:

- **Google Street View images**, one per building, named by address (scraped with Selenium)
- A corresponding **CSV file with building characteristics**, such as:
  - Exterior finish (e.g., brick, stucco, frame)
  - Built year
  - Number of stories
  - Roof type, basement presence, etc.

Since I already have image paths and metadata, I believe I can quickly relabel or clean this dataset.
This approach will:

- Align with my master's thesis direction
- Give me control over the quality, structure, and labels
- Open up future use for multi-task learning, transfer learning, and potential expansion thanks to the additional semantic data available

Analysis Plan:

1. **Preprocessing**
   - Clean images for obstructions (trees, cars) and normalize brightness/contrast (?)
   - Resize and standardize images for CNN input (?)
   - Augment for generalizability (flip, crop, brightness jitter) (?)
2. **Modeling**
   - Use a pre-trained CNN (e.g., ResNet50 or EfficientNet) with one branch:
     - classification of facade material

- Experiment with fine-tuning and regularization

3. **Evaluation**
   - Classification: Accuracy, F1-score, confusion matrix (?)
   - Regression: RMSE, MAE, R^2?
   - Visualize attention maps for explainability (?)

Caveats & Future Considerations:

- Since Google Street View images are **raw and unprocessed**, I may need to develop a pipeline to handle:
  - Brightness variation
  - Shadows
  - Image obstructions (trees, parked vehicles), etc.

I am excited since this project can serve as a foundation for my master's thesis, where I plan to explore how vision models can automate urban-scale building stock classification for urban building energy models (UBEM) and retrofits.