**Report**

1. In this project, we focused on simple Multi-Layer Perceptron (MLP) architectures with 2-3 hidden layers, considering the addition of Dropout and Batch/Layer Normalization for regularization and stability. Our primary goal was to achieve high accuracy while maintaining simplicity and efficiency in the architecture, given the nature of the dataset.

**Reasons for Focusing on MLPs:**

1. **Initial Baseline Results:**

We started by using the default classifier provided, which included only one linear layer. After training with the Adam optimizer and using the ReduceLROnPlateau scheduler, we achieved 82% validation accuracy. This indicated that the data was largely linearly separable.

Given this result, we concluded that more complex architectures might not be necessary for this task, as a single layer was already capturing a significant portion of the decision boundaries.

1. **Supporting Literature:**

We reviewed two papers [1, 2] that applied Implicit Neural Representations (INRs) as input to neural networks for classification tasks. Both papers used simple MLP architectures and reported competitive results compared to state-of-the-art (SOTA) models.

This further motivated our decision to stick with MLPs, as the results in the literature showed that MLPs were effective for classification tasks on datasets like ours.

1. **Suitability of Other Architectures:**

Convolutional Neural Networks (CNNs): We considered whether CNNs might improve performance but concluded that they would not be appropriate for our data. Each sample in our dataset is a dense 512-dimensional vector that has already been trained on the original image and contains embedded information. Convolutions are typically used to extract spatial features from raw image data, and since our input already contains highly abstracted information, CNNs would likely perform similarly to linear layers.

Transformers and Attention: We also ruled out using transformer architectures or attention mechanisms because the data does not lend itself to the standard tokenization process used in transformers. Since each sample is a dense vector and cannot be split into patches or tokens, we concluded that attention mechanisms would not provide any significant advantages for this task.

**Our final architecture**

* Input Layer: 512 → 512 (Fully Connected) + BatchNorm + LeakyReLU + Dropout.
* Hidden Layer 1: 512 → 256 (Fully Connected) + BatchNorm + LeakyReLU + Dropout.
* Hidden Layer 2: 256 → 128 (Fully Connected) + BatchNorm + LeakyReLU + Dropout.
* Output Layer: 128 → 10 (Fully Connected)

Optimizer- AdamW with learning rate=0.001 and weight decay=0.01.

Scheduler – ReduceLROnPlateau with factor=0.1 and patience =3. The scheduler uses validation accuracy as metric.

Dropout probability- 0.1

Batch size- 64

Early stopping mechanism that stops after 5 iterations with no improvement.

The batch normalization layer uses momentum=0.05 which is more suitable for smaller batch.

Checkpoint mechanism to save the set of weights that resulted the highest accuracy on validation set.

Our final network architecture was a good choice because it provides a good balance between simplicity and performance. We used a simple MLP with 4 fully connected layers, gradually reducing the dimensions from 512 → 256 → 128 → 10.

The gradual reduction in dimensionality allows the network to progressively distill the most **Batch Normalization** stabilized training by normalizing activations, and **LeakyReLU** prevented the "dying ReLU" problem, ensuring the neurons remained active throughout training.

**Batch Normalization** stabilized training by normalizing activations, and **LeakyReLU** prevented the "dying ReLU" problem, ensuring the neurons remained active throughout training. The **ReduceLROnPlateau** scheduler helped the model fine-tune after the initial stable learning phase, reducing the learning rate to avoid oscillations or overfitting. Combined with **Dropout** and **weight decay** in the **AdamW** optimizer, overfitting was well controlled. The **early stopping** mechanism ensured that we didn't overtrain the model, stopping when there was no improvement after 5 epochs.

As we’ll explain in the next section, hyperparameter tuning was a critical part of arriving at this final design. The chosen architecture achieved the highest validation accuracy during training at **89.66%**.

In addition, we implemented a checkpoint mechanism that selects the model with the highest validation accuracy during training. After training is completed, the model is reloaded with the best weights.

**Hyper parameter tuning:**

We performed hyperparameter tuning on the following parameters:

* Learning rate: [0.001, 0.0001]
* Hidden dimensions: [ [512, 256, 128], [256, 128], [128, 128], [512,512,512], [128,64] ]
* Dropout probability: [0,0.1,0.3]
* Batch size: [32,64,128,256]
* Normalization after linear layer: [None, layer normalization, batch normalization]
* Activation: [Relu, LeakyRelu]
* Early Stopping:[5,10]

There were 1,440 possible combinations, which was too large for comprehensive training. To handle this, we used Random Search, where we randomly selected 150 combinations to train and evaluate. Each combination was trained on the training set and evaluated on the validation set. We ultimately selected the combination that achieved the highest accuracy on the validation set (as described in the final architecture section).

The maximum number of epochs was set to 500, and this parameter was not tuned. Prior to the random search, we manually experimented with different optimizers and schedulers, but none performed as well as the chosen ones, so we did not tune the optimizer or scheduler further.

We decided to use AdamW, inspired by [2] and [3], as it provided more stable results compared to the standard Adam optimizer, especially in terms of generalization and convergence.

**Explanation of the choice of parameters**-

Depth+number of neourons:

We considered several options for the depth and hidden dimensions of the neural network. Inspired by [2], we aimed to test MLP architectures with reduced dimensions in each consecutive layer. Additionally, we were interested in architectures with consistent dimensions in each hidden layer, inspired by [1].

Initially, we manually tested a few MLP architectures and observed that networks with 3-4 total layers provided the best performance. Adding more than 4 layers did not lead to any noticeable performance improvement. Based on this, we concluded that the most effective approach would be to focus on MLPs with 3-4 layers during hyperparameter tuning.

Architectures Tested During Tuning:

Reduced dimensions in each consecutive layer architectures:

* (512->512) -(512->256)-(256,128)-(128,10)
* (512->256) -(256,128)-(128,10)
* (512->128)-(128,64) -(128,10)

We chose these architectures to explore how a gradual reduction in neuron count across layers impacts performance. Our goal was to maintain enough neurons in the early layers to capture important features, while progressively reducing dimensions to distill the key information. When manually testing architectures with **more than 512 neurons**, we saw no improvement, and networks with fewer than **64 neurons** didn't perform well enough.

Same size Layers architectures:

* (512->512)-(512->512)-(512->512)- (512->512)
* (512->128)-(128,128) -(128,10)

These architectures were chosen to see if consistency across layers could improve performance by maintaining a more uniform learning process throughout the network. These architectures were inspired by [1]. The specific sizes of 128 and 512 were chosen because we wanted to try the 128 size, same as in [1], and we also wanted to mimic the 4-layer architecture used in [1] but with our input size (512).

Normalization:

We aimed to explore whether normalization applied after the linear layer could improve network performance and accelerate training. Since there wasn't much information available on whether normalization would be beneficial for the INRs dataset, we decided to test whether normalization was necessary at all, and if so, whether Layer Normalization or Batch Normalization would be more effective.

Activations:

Before conducting hyperparameter tuning, we manually tested various activation functions, including Sigmoid, SiLU, ELU, GELU, ReLU, and Leaky ReLU. However, during these experiments, we observed that only ReLU and Leaky ReLUconsistently provided good performance in terms of both accuracy and training stability.

Based on these observations, we decided to focus our hyperparameter tuning specifically on ReLU and Leaky ReLU, as they proved to be the most effective for this task.

Batch size:

We decided to consider [32, 64, 128, 256] as possible batch sizes during hyperparameter tuning. Before this, we manually tested larger batch sizes like 512 and smaller batch sizes like 16, but observed that the performance was suboptimal, and the training was unstable. Larger batch sizes led to slower convergence and reduced generalization, while smaller batch sizes introduced too much noise in the gradients, leading to instability. Based on these preliminary tests, we focused on the more balanced range of 32 to 256 for tuning.

Learning Rate:

We chose to explore two learning rates, 0.001 and 0.0001, during hyperparameter tuning. Based on initial experiments, we found that larger learning rates (greater than 0.001) caused the model to oscillate and fail to converge, while smaller learning rates (below 0.0001) resulted in very slow progress.

Dropout:

We tested [0, 0.1, 0.3] for dropout during hyperparameter tuning. 0 (no dropout) was included to see if dropout was necessary. The values were chosen based on preliminary manual tests.

Early Stopping:

We considered [5, 10] for the early stopping patience. Early stopping is used to halt training if the validation performance doesn’t improve after a certain number of epochs, preventing overfitting and saving computational resources. A patience of 5 epochs was chosen for faster stopping if no improvement was seen, while 10 epochs allowed for more training before halting, in case the model needed extra time to stabilize. This gave us flexibility in balancing between early termination and giving the model enough time to improve.

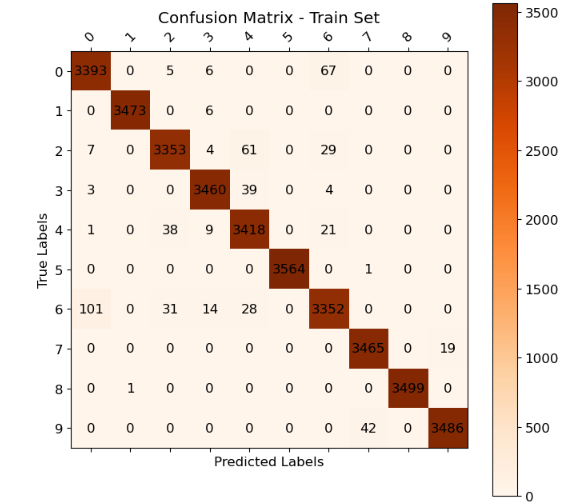
We didn’t choose a value less than 5 because our scheduler patience is set to 3, meaning the learning rate is reduced after 3 epochs without improvement. By allowing at least 5 epochs for early stopping, we ensure that the effect of the reduced learning rate can be explored before halting the training. This prevents premature stopping and allows the model a chance to improve after the learning rate adjustment.

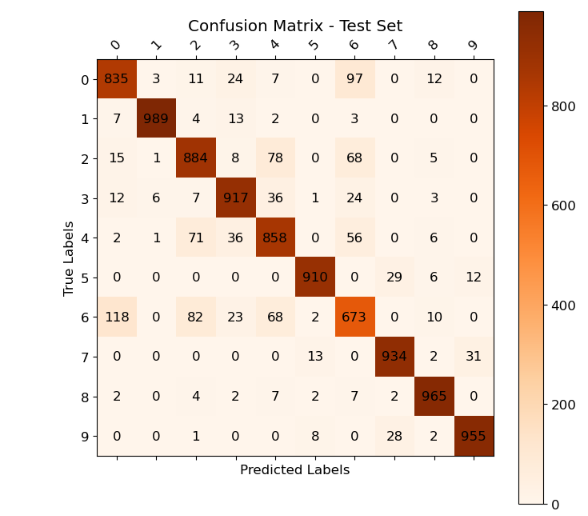
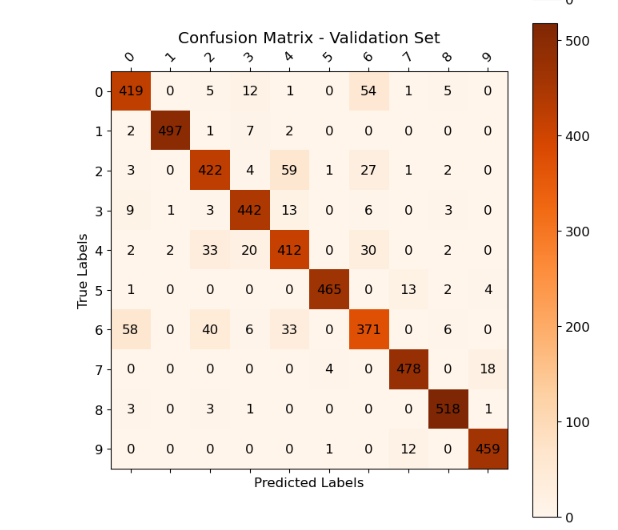
2)

Train Accuracy: 98.47%

Validation Accuracy: 89.66%

Test Accuracy: 89.10%





4) We can draw several key insights regarding the performance of the classifier across these datasets:

* The dominant diagonal across all three confusion matrices reflects that the majority of samples were correctly classified in their respective categories. This indicates that the model generally performs well across the datasets.

However, the diagonal becomes less prominent, or lighter, in the validation and test sets compared to the training set. This suggests potential overfitting.

* Major Confusions:

**Class 0 (T-shirt/top) vs. Class 6 (Shirt):** There is a confusion between these two classes, with the model often predicting class 0 as class 6, and vice versa. This bidirectional confusion indicates that the features distinguishing these two classes are not well captured by the model.

**Class 6 (Shirt), Class 4 (Coat), and Class 2 (Pullover):** There is also a confusion among these classes(in all directions). This may suggest that the model struggles with identifying visual differences between these types of clothing, possibly due to similarities in the type.

**Class 5 (Sandal) vs. Class 7 (Sneaker) and Class 7 (Sneaker) vs. Class 9 (Ankle Boot):** These confusions appear primarily in the validation and test sets, highlighting that the model may struggle with distinguishing between different types of footwear.

* The training set confusion matrix is more strictly "diagonal" compared to the validation and test sets, with fewer misclassifications. The presence of more off-diagonal elements in the validation and test sets indicates that the model has slightly overfitted to the training data.

For example, some confusions, such as misclassifying class 5 (Sandal) as class 9 (Ankle Boot), are not present in the training set but appear in both the validation and test sets.

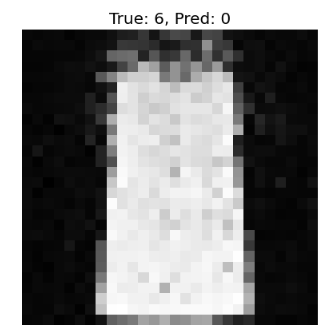
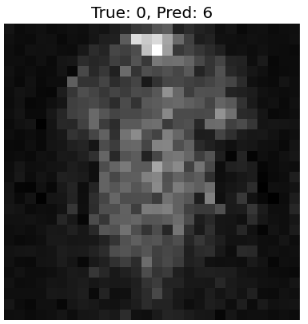
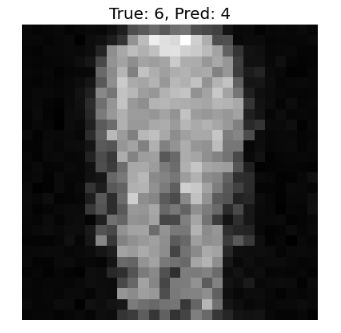
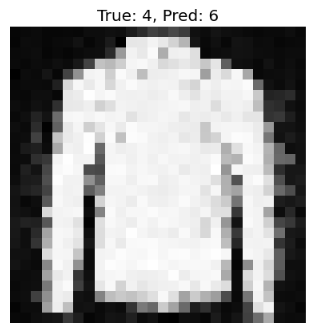
* The confusion matrix for the validation set serves as a good indicator of the model's performance on the test set, as most of the misclassifications observed in the test set are also present in the validation set.

However, there are additional confusions in the test set that are not observed in the validation set. For example, class 8 (Bag) is confused with multiple other classes (classes 2-7) in the test set, whereas in the validation set, confusion involving class 8 is more limited (mainly between classes 2 and 3).

**Improvements if we would retrain/fine tune:**

* **Filtering Misrepresented Examples:**

During the analysis, we visualized several misclassified examples using the reconstructed images from the INRs:

****

**b**

As we can observe, Some of the reconstructed images are so poor that even a human eye would struggle to identify the original clothing item. This suggests that the INR representation might not be capturing the key features needed to properly reflect the original image. Since we can’t recreate or modify the INR, one idea to improve performance would be to filter out examples where the INR’s reconstruction is clearly flawed. We could look at the reconstructions for each class, identify the ones that lead to frequent misclassifications, and consider removing those examples from the training set.

That said, not all misclassified images can be blamed on poor reconstructions. In some cases, like the example on the right, the reconstruction actually looks quite good, which suggests that the misclassification might be due to other factors.  **2) Re-weighting Classes in Cross-Entropy Loss:**

Another improvement would involve re-weighting specific classes in the CrossEntropyLoss function. As observed in the confusion matrices, the classifier tends to confuse certain classes more frequently than others—particularly class 0 (T-shirt), class 6 (Shirt), class 2 (Pullover), and class 4 (Coat). Given the higher frequency of confusion in these classes, increasing the weight for these classes in the loss function would give the model a stronger signal to classify them correctly.

The PyTorch CrossEntropyLoss function includes a weight parameter that allows re-weighting the importance of different classes, and we would experiment with different weight configurations to see if this helps reduce confusions for the problematic classes.

**3) Reducing Overfitting:**

The confusion matrices suggest some degree of overfitting, with the model performing slightly better on the training set compared to the validation and test sets. To combat this, we could increase the dropout rate to introduce more regularization, helping the model generalize better to unseen data. Additionally, we could increase the weight decay (L2 regularization) parameter in the AdamW optimizer. This would penalize large weights, encouraging the model to avoid overfitting. Hyperparameter tuning could also be expanded to include the weight decay parameter, which we haven't fully explored before.

**4) Reweightning samples**

We could try reweighting some of the samples where the model consistently makes mistakes. One way to do this, for example, is by using a weighted sampler in the DataLoader, which would sample these problematic examples more frequently during training. By giving these harder samples more focus, we hope the model would learn to handle them better, potentially reducing the overall misclassification rate.

**10. Bonus**

The approach we chose for training the classifier is adversarial training using the PGD (Projected Gradient Descent) attack to generate adversarial examples.

This approach was introduced in [4].

The inspiration for this approach comes from robust optimization, where a typical optimization problem is formulated as: .

The goal is to find a solution x that minimizes the objective function under the worst realization of u.

In the context of adversarial robustness, this framework can be applied to model training by considering perturbation attacks.

The optimization objective suggested in the paper was:

The inner maximization problem aims to find an adversarial version of a given data point x that achieves a high loss. This is precisely the problem of attacking a given neural network. On the other hand, the goal of the outer minimization problem is to find model parameters so that the “adversarial loss” given by the inner attack problem is minimized. This is precisely the problem of training a robust classifier using adversarial training techniques.

The perturbation that maximizes to create the adversarial example is generated by PGD attack, starting from random restart points within the ball around the original x.

This approach encourages robustness because the problem's formulation ensures that if we achieve a small robust loss, no allowed attack will be able to fool the model. This is because the model becomes resistant even to the worst-case adversarial attack.

However, the paper notes that solving this saddle point problem is challenging due to the non-convex nature of the outer minimization problem and the non-concave nature of the inner maximization problem. Despite this, the authors show empirically that the problem can be effectively addressed in practice by using PGD to solve the inner maximization, yielding strong robust results.

On a more intuitive level, we believe this method promotes robustness because the model is continuously exposed to adversarial examples during training. These examples push the model to learn more complex decision boundaries that are better equipped to handle subtle changes in input.

This approach has been widely adopted in the field of adversarial training.

After this paper was published, other research suggested that the formulation of the problem leads to an inherent trade-off between accuracy and robustness [5].

This has inspired the development of hybrid formulations, such as the one proposed in [6]:

In this formulation, is the adversarial example generated by the same PGD attack as before.

We implemented adversarial training in training.py, inspired by the code from [7], and the attack is implemented in attack.py. Since our classifier architecture includes batch normalization layers, the mean and variance statistics of these layers are affected by every forward pass.

To address this, we followed the approach suggested in [8]: the model is put into evaluation mode during the attack generation to prevent batch normalization statistics from being updated by the adversarial examples. Additionally, we process the model on a concatenated batch of clean and adversarial examples, ensuring that batch normalization statistics are updated based on both types of inputs in each iteration. We found that computing the model separately on clean and adversarial examples caused instability in the batch normalization layers, as the statistics of the two groups are often different, leading to inconsistent behavior.

We used the same original classifier with the same hyperparameters as described in the first part of the project. Instead of using a specific optimizer for generating adversarial examples, we manually implemented the gradient step. For adversarial example generation, we chose to use 10 pgd steps, bound=0.01, step size = bound/4, where represents the weight in the hybrid approach.

Additionally, we did not use random restarts in generating adversarial examples because we observed that it led to less improvement in robustness.

All parameters were chosen through manual experimentation, testing different combinations to find a good balance between accuracy and adversarial robustness.Initially, we experimented with larger bounds, expecting better robustness, but found that too large of a bound actually reduced robustness. This may suggest that overly large bounds fail to create subtle adversarial examples, which are necessary for creating sharper decision boundaries.

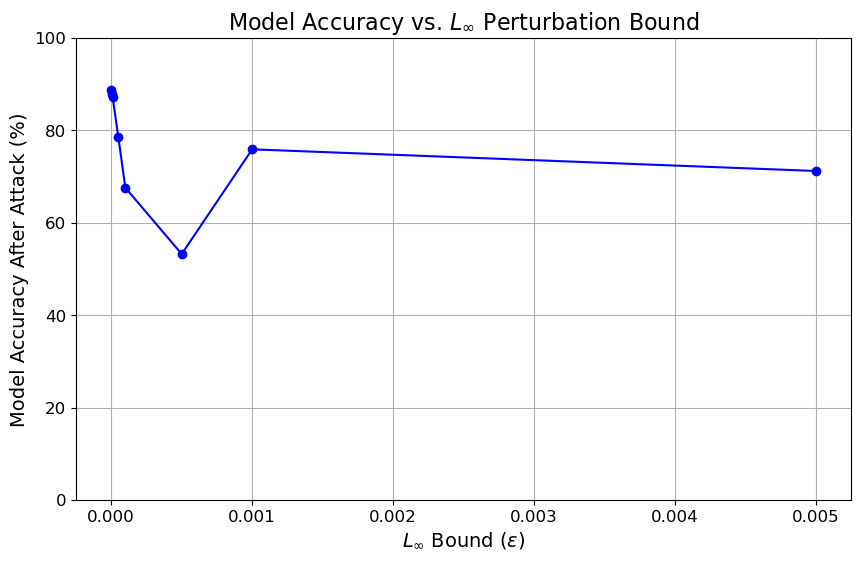
We also experimented with increasing the weight of the adversarial loss by trying different values of , but this reduced the classifier's performance to below 80% and did not result in significant improvements in robustness under attacks.

**Results:**

**Clean classification accuracy:**Train Accuracy: 95.44%

Validation Accuracy: 89.48%

Test Accuracy: 88.70%



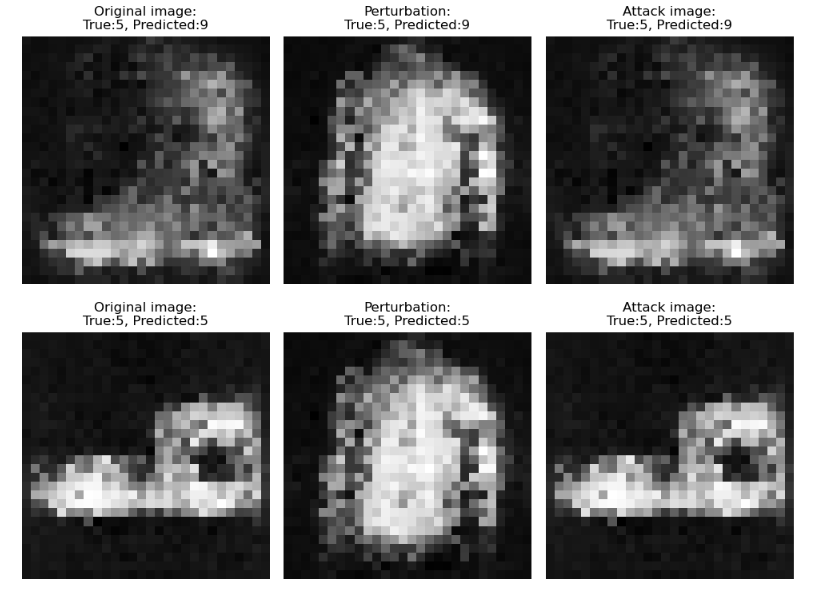
****

As observed from the results, we were able to achieve a significant improvement in the robustness metric. Compared to the classifier from the previous sections, which saw its robustness drop to zero for bounds of 0.0005 and higher, this classifier demonstrates high robustness, particularly for larger bounds.

However, we observe an interesting phenomenon: the model exhibits lower robustness for mid-sized bounds compared to larger bounds. This could be due to the fact that we trained our model using a large bound (). Training on such a large bound likely generated adversarial examples that increased the model's robustness against larger perturbations but did not sufficiently improve robustness for mid-sized perturbations.  
This might occur because the model’s decision boundaries became more focused on handling extreme cases, allowing mid-sized or smaller perturbations—which are more subtle—to still bypass the model's defenses.

To improve robustness across different perturbation sizes, it might be necessary to train on multiple perturbation sizes, ensuring the model can effectively defend against both subtle and large adversarial examples.

**Plotted images from class 5:**



**Dry part:**

1

(a)

The Universal Approximation Theorem (UAT) states that given enough parameters, a multi-layer perceptron (MLP) with one hidden layer and a non-linear activation function can approximate any continuous function on a compact subset of to arbitrary precision.

Specifically, for a continuous function

there exists a function of the form that can approximate to any given precision. Here, are parameters that adjust to the required level of precision.

This theorem supports the claim that you can achieve optimal error over any dataset and loss criterion. If there exists an optimal function that minimizes the loss on a given dataset, he UAT guarantees that there exists a single hidden layer MLP that can approximate to any desired precision. This means that such an MLP can approximate

closely enough to behave in the same way on the dataset. As a result, using an MLP with one hidden layer provides a potential pathway to achieving the minimal loss.

\*If the loss function does not have a finite minimum, we can refer to as the function that achieves the desired level of loss.

(b)

The conclusion that one should "never use more than one hidden layer" is incorrect because the UAT is simply an existence theorem. While it guarantees that an MLP with one hidden layer can approximate any continuous function, it does not provide a practical way to find this approximation. There are several reasons why using only one hidden layer might not be the best choice in practice:

* Unknown Number of Neurons: The UAT does not specify the number of neurons required in the hidden layer to achieve the desired precision. In practice, k could be extremely large, making the network impractically wide and computationally expensive.
* Optimization Challenges: The way we train neural networks (e.g., with backpropagation and gradient descent) does not guarantee finding the optimal solution. The optimization process can get stuck in local minima, saddle points, or regions with poor gradients, preventing the network from achieving the desired approximation.
* Empirical Success of Deep Networks: While the UAT applies to single hidden layer networks, deeper networks (those with more than one hidden layer) have been found empirically to perform better. They can learn hierarchical features, leading to better generalization, more efficient learning, and often better performance in real-world tasks.

3.

We would expect momentum to help in the optimization process, even for convex optimization problems. The momentum method utilizes an exponential moving average of gradients from previous steps to update the parameters in the current step. This means the current parameter is adjusted based on both the current gradient and past gradients. In convex optimization, the negative gradient consistently points toward the same global minimum. As a result, the momentum method accelerates the updates in the direction of the minimum, leading to faster convergence.  
In addition, the momentum method can smooth out updates in cases of oscillations—such as when the learning rate is too large, and the gradients change direction in each step. This smoothing effect leads to a more stable optimization process.

For example, as we learned in the course on numerical algorithms, when applying the steepest descent method to least squares problems (which are convex), using a normalized learning rate in each step can cause the gradient at each step to become perpendicular to the previous gradient. This phenomenon slows down convergence because the optimizer tends to zigzag across the loss landscape rather than moving directly toward the minimum. Introducing momentum helps combat this issue by accumulating the direction of previous gradients, allowing the optimizer to maintain a more consistent and directed path toward the global minimum.

However, it’s important to note that momentum is not always superior to standard gradient descent. If the learning rate or momentum factor is not well-tuned, it can lead to overshooting or slower convergence.

5.

Given that the kernel size has dimensions 32x32 (the same as the dimensions of the input feature map), this convolutional layer is equivalent to a **linear layer with one neuron.** The output of the model is a scalar.

Let:

- The kernel be:

- The input be:

- The bias is a scalar .

The model output is given by:

.

Since the loss computes the difference between an y, and is a scalar while y is a 32x32 matrix, broadcasting is applied. Broadcasting repeats the scalar ​ over all cells of the matrix y during the subtraction.

The loss term is therefore:

Using the chain rule :

7

(a)

Each pixel in the map represents the attention weight between a word in the source sequence (English) and a word in the target sequence (French). The attention weight shows how much influence the English token has when creating the context-based representation for the French token by applying the attention weights. The attention weight is based on the similarity between the query (target token) and the keys (source tokens). A higher weight means the English token played a more significant role in forming the representation of the French token. The color of the pixel represents the strength of attention: lighter colors indicate stronger relationships (higher attention weights), while darker colors indicate weaker relationships.

(b)   
Rows with only one non-zero pixel mean that the attention for that particular French word is focused entirely on a single English word. This typically occurs when the model identifies a direct, straightforward translation between the French and English tokens. In this case, the model does not need to distribute attention across multiple words and has clearly identified that the English token is the most relevant for translating the French token. As shown in the image, these single-focus attention rows often correspond to one-to-one word translations between the two languages.

(c)

Rows with several non-zero pixels mean that the attention for that particular French word is distributed across multiple English tokens. This indicates that the translation of the word is not a straightforward one-to-one relationship, or that more context is needed to understand the French token. For example, in the case of "L'", the attention is distributed between the matching word in English ("the") and the objects it refers to, such as "agreement" and "signed". This shows that the model is using information from multiple English words to form the correct French translation.

(d)  
 The attention weights for each French token must sum to 1 (the weights are normalized during calculation using softmax). When a row has only one non-zero pixel, that single token receives all the attention (with a weight of 1), which is why the pixel appears white, indicating maximum attention. In contrast, when the attention is distributed across multiple tokens in the source sequence, each token is assigned a fraction of the total attention, with individual weights less than 1. Since the attention is divided, these pixels will have lower values and therefore appear gray rather than white.

9.

We saw that , meaning the log-likelihood of is constructed from a KL-divergence term and an ELBO term. The KL divergence is always non-negative. Therefore, we can conclude that . This means the ELBO serves as a lower bound for .

Since our goal is to maximize (in line with the maximum likelihood estimation approach), if we maximize the ELBO term, we also maximize what we want to optimize: the likelihood.

1. To compute the KL-divergence between and , we can either estimate it empirically (by sampling from the distributions) or use an analytical solution if the distributions are explicitly defined. In this case, we don’t have an exact form for these distributions.

While we could attempt to approximateusing Markov rules, this would require estimating intermediate values , which is generally avoided during training. Instead, we typically compute directly , as it's more practical.

On the other hand, is estimated through a neural network. While we could use this neural network to estimate the full distribution ,it becomes challenging because we would need additional unknown values, such as the joint distribution or marginal probabilities like

Moreover, even if we had these distributions**, computing the expectation required for the KL divergence itself can be difficult.** The expectation involves integrating over all possible sequences, which can be computationally expensive or intractable, particularly when working with high-dimensional distributions

**Another answer:**  models the probability of a sequence given the final image This is hard to compute since models the reverse process .

Even if we use formulas like we will still stuck with problematic terms we don’t have like (which is essentially what we try to compute).

1. We ignore this term because it is constant with respect to the learned parameters

Since is not learned and is a known prior, this KL-divergence does not depend on . Therefore, its gradient is zero, meaning it does not affect the optimization process, allowing us to safely ignore it during training.

**Bibliography**

[1] Emilien Dupont, Hyunjik Kim, S. M. Ali Eslami, Danilo Jimenez Rezende, and Dan Rosenbaum. From data to functa: Your data point is a function and you can treat it like one. In International Conference on Machine Learning, pp. 5694–5725. PMLR, 2022.

[2] L. De Luigi, A. Cardace, R. Spezialetti, P. Zama Ramirez, S. Salti, and L. Di Stefano. Deep learning on implicit neural representations of shapes. In International Conference on Learning Representations (ICLR), 2023.

[3] Kim, T. Generalizing MLPs with dropouts, batch normalization, and skip connections. arXiv preprint arXiv:2108.08186, 2021.

[4] Madry, A., Makelov, A., Schmidt, L., Tsipras, D., & Vladu, A. Towards deep learning models resistant to adversarial attacks. In International Conference on Learning Representations (ICLR), 2018.

[5] Zhang, H., Yu, Y., Jiao, J., Xing, E. P., Ghaoui, L. E., & Jordan, M. I. Theoretically principled trade-off between robustness and accuracy. In International Conference on Machine Learning (ICML), pp. 7472–7482, 2019.

[6] Haotao Wang, Aston Zhang, Shuai Zheng, Xingjian Shi, Mu Li, & Zhangyang Wang. Removing batch normalization boosts adversarial training. In ICML 2022, pp. 23433–23445. PMLR, 2022.

[7] Adversarial Machine Learning Tutorial. (n.d.). Adversarial Training. Retrieved October 22, 2024, from <https://adversarial-ml-tutorial.org/adversarial_training/>

[8] Stutz, D. (n.d.). 47.9% robust test error on CIFAR10 with adversarial training and PyTorch. Retrieved October 22, 2024, from <https://davidstutz.de/47-9-robust-test-error-on-cifar10-with-adversarial-training-and-pytorch/>