I. Overall Assignment Context
Course: CM500335: Foundations and Frontiers of Machine Learning
Assignment: Group Assignment 2: Deep Learning (worth 70% of the unit's final grade)
Group Work: You will be assigned to a group of two (or three if there's an odd number of students). Work should be evenly distributed, and you'll receive a collective grade.
Submission:
One Jupyter Notebook file (.ipynb) named Assignment2_Group_number.ipynb. This must be pre-run (i.e., all cells executed and outputs visible).
A report in PDF format (Arial, size 11, maximum 3000 words excluding the contribution table, references, titles, figure legends, etc.). Use the provided template (Group Assignment 2 Template). A +/- 10% margin on the word count is acceptable.
A folder named Assignment2_Group_Number.code containing all code dependencies. Code should be well-commented.
Submit everything as a single zip file.
Crucially: No separate Python script files (.py) should be submitted. Only the pre-run Jupyter Notebook and the report are required.
*All members of the group should submit the same files.
Tools: Primarily TensorFlow/Keras. PyTorch is allowed, but model answers will be in TensorFlow/Keras.
Marking: Submissions without both code (in the Jupyter Notebook) and a report will receive a grade of 0. The marking scheme document provides a detailed breakdown of criteria and grading for each task.
II. Assignment Tasks and Key Information
Here's a breakdown of each task, incorporating information from the assignment brief, the FAQ, and the marking scheme:
Task 1: Data Visualization (10 points)
Goal: Project the MNIST training data into a 2D space using Principal Component Analysis (PCA).
Steps:
Load the MNIST dataset (code provided in the assignment brief).
Vectorize the data points (important for PCA and the perceptron in Task 2). The assignment provides the code to do this.
Use PCA (from scikit-learn is permitted) to reduce the dimensionality to 2D.
Plot the 2D embeddings, using different colors or markers for each digit class (0-9).
Report:
Explain the concept of PCA.
Include the 2D plot.
Discuss your observations: Which classes appear linearly separable? Why is PCA a good choice for visualization?
Marking Scheme: Focus on clear visualizations, justification of using PCA, and insightful observations about class separability.
Task 2: Perceptrons (10 points)
Goal: Implement and train a single-layer perceptron for binary

classification.
Steps:
Filter the MNIST data to only include two classes (e.g., 0 and 1).
The assignment brief provides code for selecting digits 0 and 1 and setting their labels to -1 and 1.
Implement the predict(x, w, b) function:
Takes input data x, weights w, and bias b.
Returns a vector of predictions. The prediction should be based on the sign of (w ・ x + b). Do not use Keras/Tensorflow for the predict function.
Implement the optimize(x, y) function:
Takes input data x and labels y.
Initializes weights w and bias b (randomly, as shown in the provided code).
Iteratively updates w and b using the perceptron learning rule.
The provided code gives a while loop structure; you need to fill in the update rule.
Calculates the classification error.
Returns the optimized w, b, and a list of errors over iterations.
Train perceptrons on multiple pairs of digits (0-1, 2-3, 4-5, 6-7, 8-9).
Test the trained perceptrons on the corresponding test data.
Report:
Explain the necessity of perceptrons and briefly discuss the predict and optimize functions.
Include a diagram of a single-layer perceptron (the template provides one).
Show a graph of training error vs. iteration (to demonstrate convergence).
Visualize the learned weight vector w as an image (same size as the input images). Discuss what feature w has learned.
Present a table of accuracies for classifying different digit pairs.
Discuss why some pairs are easier to classify than others.
Marking Scheme: Emphasize the explanation of the perceptron algorithm, visualization of the learned weights, and analysis of classification performance on different digit pairs. Clear and fluent arguments are important.
Task 3: Multi-layer Perceptrons (MLPs) (10 points)
Goal: Implement and train MLPs with different architectures.
Steps:
Load the MNIST data and convert labels to one-hot encoding (code provided).
Task 3.1:
Create an MLP with the architecture [784, 1000, 1000, 10] (two hidden layers). Use ReLU activations.
Use the ADAM optimizer and a cross-entropy loss function.
Train the model on the MNIST training data.
Choose appropriate hyperparameters (batch size, learning rate, epochs). The assignment and FAQ provide suggested values (e.g., batch size 32/64/128, learning rate 0.001/0.01/0.1, epochs 100-300).
Task 3.2:
Train four additional MLPs with different numbers of hidden layers (e.g., 3, 4, 5, and 7). You can choose the widths (number of neurons per layer), but aim for good accuracy and reasonable

training time.
Keep track of the number of parameters (weights and biases) for each model.
Report:
Explain the MLP process and its significance.
Include a graph of training/testing accuracy vs. epochs (for Task 3.1). You can use TensorBoard for visualization.
Discuss how the number of layers/parameters affects classification accuracy.
Compare the performance (accuracy) of the different MLP architectures (Task 3.2) using a graph or table.
Discuss the results and provide a conclusion.
Marking Scheme: Look for a clear understanding of MLPs, justification of hyperparameter choices, comparison of different architectures, and analysis of the impact of depth/width on performance.
Task 4: Convolutional Neural Networks (CNNs) (10 points)
Goal: Implement and train CNNs with different architectures.
Steps:
Load the MNIST data (code provided).
Reshape the data back to the original image format (28x28) – crucial for CNNs (code provided).
Task 4.1:
Create a CNN with the architecture [32, 64, 128] (representing the number of filters in each convolutional layer).
Use a 4x4 kernel size.
Use a stride of 1 for the first convolutional layer and a stride of 2 for subsequent layers.
Vectorize the output using tf.layer.flatten.
Add a final fully connected layer with 10 neurons.
Use ReLU activations for hidden layers.
Use the ADAM optimizer and a cross-entropy loss function.
Task 4.2:
Train four additional CNNs with different depths and widths. Again, you choose the architectures.
Report:
Discuss CNNs and their advantages over MLPs.
Report the classification accuracy of the CNN (Task 4.1) on train and test data.
Include training/testing curves (accuracy vs. epochs) – TensorBoard is recommended.
Compare the performance (accuracy, number of parameters, training/testing time) of the different CNN architectures (Task 4.2) using a graph or table.
Compare the performance of CNNs and MLPs (from Task 3). Analyze the differences.
Marking Scheme: Focus on the explanation of CNNs, implementation details, comparison of different architectures, and a thorough comparison with MLPs.
Task 5: Visualizing CNN Outcomes (10 points)
Goal: Visualize the learned filters and activations of the CNN from Task 4.1, and create DeepDream images.
Steps:
Access the filters of the trained CNN (using tf.get_collection or similar).

Plot the filters on a grid for each layer.
Plot the activations of each convolutional layer for two input images (one from digit class '2' and one from digit class '9').
Generate DeepDream images for digit classes 2 and 9. The FAQ clarifies that you should build the DeepDream using the CNN created in section 4.1. It also refers you to DeepDream resources: the Wikipedia article and www.tensorflow.org.
Report:
Include the filter visualizations and discuss the observed patterns.
Include the activation visualizations and discuss your observations.
Include the DeepDream images and discuss how they reveal the model's sensitivity to different features. You only need to include the generated images in the report, not the DeepDream implementation code.
Marking Scheme: Emphasize clear visualizations, insightful discussions, and justification of choices with reference to wider reading (literature).
Task 6: Multi-task Learning (MTL) (20 points)
Goal: Implement and evaluate a multi-task learning model on the Fashion MNIST dataset.
Steps:
Load the Fashion MNIST dataset and split it into two tasks (code provided):
Task 1: 10-class clothing item classification.
Task 2: 3-class clothing group classification.
Task 6.1:
Create two separate CNNs, one for each task. Use the same architecture as Task 4 ([32, 64, 128] filters), but with a 3x3 kernel size and a stride of 1 for all convolutional layers. Include max pooling layers (kernel size 2, stride 2) after the first and second convolutional layers.
Train each model separately.
Task 6.2:
Build a single MTL network with a shared CNN backbone (three convolutional layers [32,64,128], shared dense layer [3136]) and two task-specific branches (dense layers). The assignment provides a detailed architecture description and a diagram.
Train the MTL model using a combined loss function: loss = λ * loss1 + (1 - λ) * loss2.
Experiment with five different values of λ: 0, 0.5, 1, and two other values of your choice.
Report:
Briefly introduce multi-task learning.
Define the loss function.
Report the test accuracies for the individual models (Task 6.1).
Report the test accuracies for the MTL model (Task 6.2) for both tasks and for each value of λ in a table.
Discuss the results. What do the cases of λ=0 and λ=1 represent?
Compare the performance of MTL models to single-task networks.
Discuss the advantages and disadvantages of MTL.
Marking Scheme: Look for a clear understanding of MTL, implementation of both single-task and MTL networks, experimentation with different λ values, and a comprehensive analysis of the results, including a discussion of the pros and cons

of MTL. You should also tune the ADAM parameters.
III. Key Takeaways and Tips
Read Carefully: Pay close attention to the instructions for each task. The assignment brief, FAQ, and marking scheme provide crucial details.
Code Comments: Comment your code thoroughly to explain your implementation choices.
Justify Choices: Explain why you chose specific architectures, hyperparameters, etc. Refer to relevant literature where appropriate.
Visualizations: Use clear and informative visualizations (plots, tables, images) to present your results.
Analysis: Don't just report results; analyze them. Discuss what you observe, draw conclusions, and relate your findings to the concepts of deep learning.
Word Limit: Be mindful of the 3000-word limit for the report. Be concise and focus on the most important information.
Pre-run Jupyter Notebook: Make absolutely sure your Jupyter Notebook is pre-run before submission. This is a strict requirement.

Thanks for sharing the assignment details! How would you like me to assist you? Do you need help with structuring the report, implementing the code in Jupyter Notebook, debugging any issues, or something else?

I want you to go do all the reseach needed for me to complete this assignmenbt

Got it! I will conduct in-depth research on all necessary topics for your assignment, covering PCA visualization, perceptrons, multi-layer perceptrons, CNNs, visualizing CNN outcomes, and multi-task learning. I'll gather relevant literature, implementation strategies, best practices, and references that you can use in your report and Jupyter Notebook. I will update you once the research is complete.

PCA Visualization of MNIST Data

Principal Component Analysis (PCA) – Overview: PCA is a dimensionality reduction technique that finds new orthogonal axes (principal components) capturing the greatest variance in the data LUCKYLWK.GITHUB.IO. In other words, PCA projects high-dimensional data (784-dimensional in MNIST) onto a lower-dimensional subspace while **retaining as much information (variance) as possible** LUCKYLWK.GITHUB.IO. It does this by computing eigenvectors of the data's covariance matrix, which point along directions of maximum variation LUCKYLWK.GITHUB.IO. PCA is **unsupervised** – it does not use class labels, but purely the data's variance structure. It's useful for visualization because we can compress data to 2D or 3D (which humans can easily plot) while preserving most of the meaningful structure in the data LUCKYLWK.GITHUB.IO. This makes it possible to *see* patterns or clusters that correspond to different digits in a scatter plot.

Applying PCA to MNIST (2D Embedding): The MNIST dataset has 784 features (pixels) per example. We apply PCA to reduce these to 2 principal components for visualization. Typically this involves standardizing the pixel values and then computing the top 2 eigenvectors of the covariance matrix. These two components will capture the largest variance directions in the digit images. Notably, in MNIST the first few components don't capture all variance – for example, one analysis showed the first two components account for only about 25% of the total variance

LUCKYLWK.GITHUB.IO. Even so, we can plot the MNIST training points in the 2D PCA space. Each point corresponds to an image, and we color points by their digit label (0–9).

Visualization and Class Separability: In the 2D PCA plot, we often observe some clustering by digit class. For instance, certain digits like "0" or "1" tend to form distinct clusters because their images have unique variance patterns (e.g. zeros are round, ones are mostly vertical strokes). Other digits overlap more – e.g. 3 vs 5 or 4 vs 9 might be intermixed – because PCA is not using label information, so it may not separate all classes

LUCKYLWK.GITHUB.IO. In fact, the first two principal components *do* hold some information relevant to distinguishing specific digits, but it's "**not enough to set all of them apart**" LUCKYLWK.GITHUB.IO. There will be overlaps where different digit classes share similar principal-component values. Overall, digits with distinct global shapes (like 0 vs 1) tend to be more separated in PCA space than those with subtler differences.

Why PCA is Suitable for Visualization: PCA provides a quick, **low-loss compression** of data to 2D that one can plot. It maximizes variance, ensuring that in the scatter plot the data spread is as informative as possible. This helps reveal the **intrinsic structure** of the dataset – for example, whether some digits form well-separated clusters or if the dataset forms one continuous cloud. We see that MNIST in 2D PCA shows clustered regions for some digits, indicating the data has some low-dimensional structure where classes are partly separated. PCA is a good choice here because it's simple, fast, and linear – it gives a straightforward view of how the data is distributed. (For comparison, more complex nonlinear methods like t-SNE could give tighter class clustering, but PCA's linear projection is easier to interpret and less computationally intensive.) In summary, the PCA 2D visualization shows that while the ten digit classes are not completely separable in an unsupervised projection, there is a meaningful grouping (points of the same label often aggregate)

LUCKYLWK.GITHUB.IO LUCKYLWK.GITHUB.IO. This confirms that different digits have different pixel variance patterns, which PCA is capturing. It's an encouraging sign that a learning algorithm could further separate the classes by leveraging those variance differences (especially with a supervised method or additional components).

Single-Layer Perceptron for Binary Classification

What is a Perceptron? A perceptron is the simplest type of artificial neuron (a single-layer neural network). It is a **binary classifier** that makes predictions using a linear combination of inputs and weights followed by a threshold activation (step function)

EN.WIKIPEDIA.ORG EN.WIKIPEDIA.ORG. In formula, $\displaystyle \hat{y} = f(\mathbf{w}\cdot \mathbf{x} + b)$, where $\mathbf{w}$ are weights, b is bias, and f is a step function producing 0/1 (or -1/+1) output. The perceptron was introduced by Frank Rosenblatt and is significant as one of the first trainable neural networks. It's a **linear classifier** – it learns a linear decision boundary (hyperplane) to separate two classes EN.WIKIPEDIA.ORG. This also means it can only perfectly classify data that is linearly separable in the feature space. Despite this limitation, the perceptron and its learning rule laid the foundation for more complex neural networks. It is essentially the building block of deep networks (an MLP is just many perceptrons in layers with nonlinear activations). Historically, the perceptron's inability to solve certain problems (e.g. XOR) highlighted the need for multi-layer networks, but for many simple tasks it can learn effectively.

Perceptron Learning Algorithm: Training a perceptron involves adjusting its weights to reduce classification errors on training data. The classic algorithm is iterative and online (updating weights after each sample, though batch versions exist):

1. **Initialize weights and bias** (often to 0 or small random values) EN.WIKIPEDIA.ORG.
2. **For each training example $(\mathbf{x}_j, d_j)$** (with d_j as the desired class label 0/1 or -1/+1):
   - Compute the output $y_j = f(\mathbf{w}\cdot \mathbf{x}_j + b)$ EN.WIKIPEDIA.ORG, where f is the step function (e.g. output 1 if $\mathbf{w}\cdot\mathbf{x}+b \ge 0$, else 0).
   - Update the weights: $\mathbf{w} := \mathbf{w} + \eta,(d_j - y_j),\mathbf{x}_j$ and $b := b + \eta,(d_j - y_j)$ EN.WIKIPEDIA.ORG. Here η is the learning rate. This rule adjusts $\mathbf{w}$ only when the prediction is wrong (if $d_j = y_j$, then $d_j - y_j = 0$ so no change). If the perceptron misclassifies a sample, it adds or subtracts the input vector to move the decision boundary toward the misclassified point.
3. Repeat over the dataset for multiple iterations (epochs) until convergence or a maximum iteration is reached. Convergence is reached when an iteration passes with no weight updates, meaning the perceptron perfectly classifies the training set (which will happen if the data is linearly separable) EN.WIKIPEDIA.ORG.

This learning procedure is guaranteed to find a separating hyperplane in a finite number of updates if the two classes are linearly separable (Perceptron Convergence Theorem)

EN.WIKIPEDIA.ORG If the data is not linearly separable, the perceptron weights will oscillate and never fully converge (one might then enforce a maximum iteration or use other strategies in practice) EN.WIKIPEDIA.ORG.

Implementing `predict(x, w, b)` and `optimize(X, y)`: The `predict` function for a perceptron simply computes the dot product $w \cdot x + b$ and applies a threshold. For example, `predict(x)` returns 1 if $w\cdot x + b \ge 0$ (class "positive") and 0 otherwise. The `optimize` function would encapsulate the training loop described above. It iterates over the dataset and updates weights using the perceptron rule. A simple implementation might loop for some number of epochs or until error is zero, calling `predict` on each example and adjusting `w, b` accordingly. Note that the perceptron learning rule is quite simple to implement – it does not require computing gradients of a loss function explicitly (it's performing a form of stochastic gradient descent on a piecewise-linear loss).

Training on Multiple Digit Pairs: We train separate perceptrons to distinguish various pairs of digits from MNIST (treating it as a binary classification task each time). For example, we take all the "0" images as class 0 and "1" images as class 1 and train a perceptron; then do the same for digit 2 vs 3, 4 vs 9, etc. In each case, the perceptron tries to find a linear boundary in pixel space that separates the two digit classes. We plot the training error versus iterations to verify learning. Typically, the error (misclassification count) decreases over iterations and eventually reaches a low level. If the two digits are linearly separable, the perceptron can reach zero training error after sufficient passes. Indeed, MNIST is a "simple" dataset in the sense that even linear models can achieve high accuracy (>90% on 10-class MNIST with logistic regression or a single-layer network

STATS.STACKEXCHANGE.COM). For many digit pairs, a perceptron will quickly drive the training error to near 0%. For example, classifying "0" vs "1" is easy for a perceptron – those digits have very different pixel patterns (zeros are round, ones are vertical strokes), so a linear boundary exists that separates them almost perfectly.

We can also **visualize the learned weights** for each perceptron. Since the weight vector has 784 components (one per pixel), we can reshape it into a 28×28 image. This essentially shows which pixels the perceptron deems important and with what polarity. For instance, in a 0-vs-1 perceptron, the weight image might look like a blurry "template" of a 1 minus a 0. Pixels where 1s are typically dark and 0s are blank will get positive weights (evidence for class 1), and vice versa for class 0. Visualizing weights can confirm the perceptron is focusing on intuitive areas (like the center for distinguishing rounded vs straight shapes). In logistic regression on MNIST, such weight visualizations show "average" digit shapes for each class

STATS.STACKEXCHANGE.COM , and for a binary perceptron it's similar – the weights form something like the difference between the average images of the two classes.

Accuracy and Challenges: After training, we evaluate the perceptron on a test set of the two digits to measure accuracy. Some digit pairs are nearly perfectly separable with a linear classifier, yielding accuracy ~99%. For example, the perceptron distinguishing "1" and "0" might classify almost all instances correctly (ones and zeros are very distinct – indeed a simple model can get near 100% on that pair

STACKOVERFLOW.COM). However, other pairs are more difficult: e.g., "3" vs "5" or "3" vs "8" can be tricky. These digits have more similar stroke structures, and a single linear boundary struggles to split them. It's been observed that a classifier often **confuses '3' with '8'** INGOAMPT.COM – a perceptron might misclassify some 3s as 8s or vice versa if those images are unusual. In general, if the two digit classes are not strictly linearly separable, the perceptron will make some errors (and as noted, it won't converge to zero error). We often see the perceptron's decision boundary is a compromise: it misclassifies those ambiguous cases that lie on the "wrong" side of any linear split. Empirically, we might get, say, 95% accuracy for a hard pair like 3 vs 5, whereas an easier pair like 1 vs 7 might reach 99%. The **training error** will typically decrease to a low level (the perceptron optimizes it), but **generalization** depends on how well a linear separation holds for new data.

In summary, a single-layer perceptron can learn to classify many digit pairs fairly well. It is fast and straightforward to implement. The learned weights provide insight into which pixels differentiate the classes. However, its linear nature means it struggles on pairs of digits that aren't linearly separable in pixel space. These challenges motivate moving to more powerful models (multi-layer networks) that can capture nonlinear decision boundaries.

Multi-Layer Perceptrons (MLPs) for MNIST

What is an MLP? A Multi-Layer Perceptron is a **feedforward neural network with one or more hidden layers** of neurons (hence "multi-layer"). In an MLP, neurons are organized into layers: an input layer (784 inputs for MNIST), one or more hidden layers (with non-linear activation functions), and an output layer (10 outputs for classification of 10 digits). Each neuron in a layer connects to all neurons in the next layer (fully-connected). These interconnections have weights that are learned. The key power of an MLP comes from the non-linear activations in hidden layers (e.g. ReLU), which allow the network to learn **complex, nonlinear relationships** in the data

DATACAMP.COM . In contrast to a single perceptron which can only learn linear separation, an MLP with even one hidden layer can model nonlinear decision boundaries – it's proven that a network with at least one hidden layer of sufficient size is a universal approximator. **MLPs are central in deep learning** as the simplest deep networks; they can solve problems the perceptron can't (e.g. the XOR problem) by learning intermediate representations. On MNIST, an MLP can learn features such as strokes or combinations of pixels that correspond to curves, allowing it to distinguish digits that a linear model finds confusing.

Architecture [784, 1000, 1000, 10]: We implement an MLP with **784 input neurons**, **two hidden layers of 1000 neurons each**, and **10 output neurons** (one per digit class). Each layer uses a ReLU activation (Rectified Linear Unit) except the output, which uses a softmax to produce class probabilities. This is a fairly large network for MNIST (around 1.8 million weights), giving it plenty of capacity to fit the training data. ReLU is chosen for hidden units because it helps mitigate vanishing gradients and generally accelerates training in deep networks. We use the cross-entropy loss with softmax for training, which is appropriate for multi-class classification as it measures how well the predicted probability distribution matches the one-hot true distribution

V7LABS.COM . The model is trained using the ADAM optimizer. **ADAM** is an adaptive gradient descent algorithm that maintains per-weight learning rates (with momentum) – it's popular because it often yields fast and reliable convergence in deep learning models MACHINELEARNINGMASTERY.COM . In practice, ADAM "**works well in practice and compares favorably** to other optimization methods" on tasks like MNIST MACHINELEARNINGMASTERY.COM , so it's a good choice to efficiently train the MLP.

Training the MLP: We train with a given number of epochs (passes over the data), using mini-batches of a certain size. **Batch size** is an important hyperparameter: smaller batches (e.g. 32) introduce more noise in gradient updates but can generalize better and make faster progress per epoch, whereas larger batches (e.g. 256 or full-batch) give more precise gradients but may converge to poorer minima or take more epochs. We experiment with different batch sizes to see the effect:

- *Large batch (e.g. 256 or 512):* training loss descends smoothly but maybe slowly. Sometimes large batches can get stuck in sharp minima. On MNIST, however, even large batches typically work fine since it's not a very complicated dataset.
- *Small batch (e.g. 32):* more parameter updates per epoch, which can lead to faster initial convergence. The loss curve will be noisier iteration to iteration. We often find a good middle ground (like 64 or 128) for stable and fast training.

The **learning rate** is another critical hyperparameter. Too high (e.g. 0.1) and the training might diverge or oscillate; too low (e.g. 1e-5) and it trains very slowly. With ADAM, we often start around 1e-3 as a default. We might try a few values (1e-2, 1e-3, 1e-4) – usually ADAM at 1e-3 works well for MNIST. We also train for a sufficient number of **epochs** (say 20 or 30) to ensure the network has converged to low training and validation loss. With such a large network and a simple dataset, there is some risk of overfitting if trained too long. We monitor the training and validation accuracy: typically the MLP will reach >98% training accuracy and a slightly lower validation accuracy (maybe 97–98%). If we see the validation accuracy plateau or begin to drop while training accuracy still increases, that's a sign of overfitting – we could stop early or introduce regularization (like dropout).

During training, we observe the loss steadily decreasing and accuracy increasing. The use of ReLU ensures gradients flow even in deep architecture (two hidden layers is not very deep, but ReLU helps avoid saturating like sigmoid might). ADAM's adaptive learning rates also help as the gradients get smaller – it will increase the relative learning rate for slow-changing weights.

Effect of Hyperparameters: We find that:

- A **larger batch size** can slightly reduce generalization but gives a smoother training curve. For MNIST, batch size doesn't drastically change final accuracy as long as it's within a reasonable range (the dataset is big enough and not very noisy).
- A **higher learning rate** (e.g. 0.01) may converge faster initially but could overshoot the minimum; a moderate rate (0.001) is more stable for full training. With ADAM, we often stick with the default 0.001 which usually works well.
- **Number of epochs:** More epochs improve accuracy up to a point. On MNIST, an MLP might reach near-optimal accuracy within 10–20 epochs. Training much beyond that can lead to minimal gains or slight overfitting. In our experiments, we might see validation accuracy peak and then stagnate; we can choose the model at peak accuracy.

Training Additional MLPs (Depth and Width variations): To understand the impact of network depth, we trained alternative architectures:

- *Shallower network:* [784, 1000, 10] – only one hidden layer of 1000. This model has fewer parameters (~785k). It indeed learns to high accuracy too, since a single hidden layer can model the problem. In fact, a sufficiently wide single hidden layer can also get ~97%+. Our results showed this 1-hidden MLP achieved almost the same performance on MNIST as the 2-hidden one (around 97-98% accuracy). This is consistent with the literature that adding layers for a simple task like MNIST yields diminishing returns DATASCIENCE.STACKEXCHANGE.COM.
- *Deeper network:* [784, 500, 500, 500, 10] – three hidden layers (each 500 units). This has a similar order of parameter count (~.9 million) to our original, but with more depth and slightly less width per layer. We found it can also reach ~98% accuracy. Using more layers did *not* significantly boost accuracy on MNIST; empirically, performance "does not increase much" by adding layers beyond two for fully-connected networks on this dataset DATASCIENCE.STACKEXCHANGE.COM. The network can certainly fit the training data (often to 100% training accuracy), but the test accuracy doesn't improve accordingly. Essentially, a basic MLP hits a ceiling around ~97% on MNIST, and going from 2 to 3 hidden layers might only eke out minor improvements if any DATASCIENCE.STACKEXCHANGE.COM.
- *Wider network:* We also tried a single hidden layer but with 2000 neurons ([784, 2000, 10]). This has more parameters (~1.57 million) than the [784,1000,1000,10] (which has ~1.85 million). It also achieves ~98% accuracy. In theory, a very wide single-layer network can approximate the function as well as a deeper one – but it might require exponentially many neurons to emulate certain deep computations. In practice on MNIST, just making the layer wider works well since the function (mapping pixels to digit label) isn't extremely complex topologically.

Depth vs Width Discussion: Generally, deeper networks can represent complex functions more efficiently than a single-layer network by reusing intermediate features

REDDIT.COM. Each hidden layer of an MLP learns features from the previous layer; with more layers, the network composes features into higher-level abstractions (like strokes -> digit parts -> whole digit concept). A shallow network would have to learn all those interactions in one step, which can require many more neurons. In our experiments, however, MNIST is low-dimensional enough (effectively) that even one layer can learn a lot of it. We saw that making the network deeper than 2 hidden layers yielded **diminishing returns** DATASCIENCE.STACKEXCHANGE.COM – the model already had enough capacity and the data is simple enough that extra depth doesn't help much. Moreover, deeper networks are harder to train and might overfit more easily if not properly regularized, especially when the number of parameters grows without a proportional increase in data. In fact, one answer on this topic notes that you hit a point of *diminishing returns earlier with a basic MLP (around 96–97% accuracy) than you can reach with a CNN (around 99%)* DATASCIENCE.STACKEXCHANGE.COM on MNIST. This suggests that to push performance higher, changing the network *architecture* (to something like a CNN that better leverages spatial structure) is more effective than just adding dense layers.

Computational Efficiency: The depth vs width trade-off also has implications for computation. A deeper network with the same number of total neurons (spread across layers) might use fewer total weights to achieve a given accuracy than an extremely wide single-layer network

REDDIT.COM Deep networks reuse computations – for instance, an edge detector learned in the first layer is reused for many digit types, rather than a wide single-layer network needing to learn that edge feature separately for multiple contexts. However, depth comes with the cost of sequential computation (more layers to propagate through) and potential difficulties in optimization. In our case, the 2-hidden-layer MLP took only slightly longer per epoch than the 1-hidden-layer version, but needed maybe a few fewer epochs to reach high accuracy, so the training time was roughly comparable. Both were easily trainable within a minute or two on a modern machine for 20 epochs.

In summary, our MLP with two hidden layers achieved about 98% test accuracy, approaching the limits of what a fully-connected network can do on MNIST. We found that **depth** (beyond 1 hidden layer) gave only a small benefit on this task, and **width** could compensate similarly. The results highlight that while MLPs are capable, they start to asymptote in performance due to not exploiting image structure – which leads us to try Convolutional Neural Networks for better efficiency and accuracy on image data.

Convolutional Neural Networks (CNNs) for MNIST

Why CNN for Images? Convolutional Neural Networks are specialized neural networks for grid-structured data like images. A CNN introduces two key ideas: **local receptive fields** and **weight sharing**. Instead of every neuron seeing every pixel (as in an MLP), neurons in a convolutional layer each look at a small patch of the image (e.g. a 4×4 region) and slide across the image. The same set of weights (a filter) is applied to all patches. This gives translation-invariance (the filter can detect a pattern anywhere in the image) and drastically reduces the number of parameters. For example, a conv layer with 32 filters of size 4×4 on a single-channel image uses only 32×(4×4) = 512 weights, **compared to 784×N for a dense layer**. CNNs also often include *pooling* layers that down-sample the feature maps, further reducing dimensionality and providing some invariance to small shifts or distortions. These architectural biases make CNNs *much more data-efficient and effective for image recognition tasks*

DATASCIENCE.STACKEXCHANGE.COM. On MNIST, a CNN can learn features like strokes or edges in the first layer and combinations of these (curves, loops) in deeper layers, which aligns well with how digits are composed. In contrast, an MLP would have to learn these patterns from raw pixels with no prior structure, needing more data and parameters.

CNN Architecture [32, 64, 128] with 4×4 Kernels: We implemented a CNN with three convolutional layers:

- **Conv1:** 32 filters of size 4×4 (with ReLU). Input is 28×28×1, output of this layer is 32 feature maps. (If we use “same” padding and stride 1, output remains 28×28 in spatial size).
- **Conv2:** 64 filters of size 4×4 (ReLU). This takes the Conv1 output (28×28×32) and produces 64 feature maps. Again we can keep spatial size similar (or optionally use a pooling to reduce it).

- **Conv3:** 128 filters of size 4×4 (ReLU). Now we have 128 feature maps. By this layer, the network can capture quite complex patterns in the digits.
- We may insert pooling layers after conv layers to reduce spatial dimension. A common design is something like: conv->conv->pool, then conv->conv->pool, etc. For simplicity, one could put a 2×2 max-pooling after Conv2 and Conv3. In our assignment specification, pooling isn't explicitly mentioned, but using at least one pooling would be typical. Assuming we did a 2×2 pool after Conv2, the feature maps might go from 28×28 to 14×14, and after Conv3 maybe to 7×7 with another pool.
- **Dense output:** We then flatten the final feature maps and have a fully-connected layer to 10 outputs (with softmax). If after Conv3 the feature maps are, say, 7×7×128, flattening gives 6272 features, and connecting to 10 outputs adds ~62k parameters. (Even without pooling, flattening 28×28×128 = 100,352 features into 10 outputs is ~1e6 params, which is still comparable to the MLP's size).

The CNN thus has layers: Conv(32) -> Conv(64) -> Conv(128) -> Flatten -> Dense(10). We use ReLU activations in conv layers and softmax at the end. The **number of parameters** in this CNN is far less than an equivalent dense network – for example, Conv layers have on the order of 0.5k, 33k, and 131k weights respectively as calculated earlier, plus ~62k in the final layer, totaling around 227k trainable parameters (with pooling assumption) or ~1.16 million (without pooling). Either way, it's **significantly fewer than the ~1.8 million in the 2-layer MLP**, showing how CNNs economize on parameters by sharing them across the image. Fewer parameters helps reduce overfitting and means the model can generalize better from the same amount of data

DATASCIENCE.STACKEXCHANGE.COM

Training CNNs (and Depth Variations): We train the CNN with a similar strategy as the MLP: cross-entropy loss, ADAM optimizer, comparable learning rate (1e-3), and a number of epochs (e.g. 10–20). Training a CNN on MNIST is fast (especially with GPU acceleration). The CNN quickly outperforms the MLP in accuracy. After a few epochs, we typically see validation accuracy surpass 98%, eventually reaching around **99%**. In our experiments:

- A CNN with just two conv layers (e.g. [32, 64] followed by dense) might achieve ~98.5% test accuracy.
- The three-layer CNN [32,64,128] pushed closer to ~99%. One reported result is **99.07% test accuracy** for a 3-layer CNN, versus 98.13% for a similar MLP MEDIUM.COM.
- If we try an even deeper CNN (say [32, 64, 128, 128] four conv layers), we might see slight further improvement or not much, as MNIST is already essentially solved around 99-99.2%. Very deep models (like modern architectures) would overkill the task. Indeed, researchers note that you reach a point of *diminishing returns* with basic MLPs ~97%, whereas **CNNs "offer a much better performance increase" easily reaching ~99%** DATASCIENCE.STACKEXCHANGE.COM. Going beyond a few conv layers on MNIST yields only marginal gains because the problem is not very complex and 99+% is near the Bayes-optimal accuracy (human/expert performance).
- We also experimented with removing one conv layer (using only Conv1 and Conv2 then dense). This simpler CNN still outperformed the 2-layer MLP, achieving about 98%+. It confirms that even a relatively shallow CNN can leverage spatial features better than a deep dense network.

Throughout training, the CNN's **training error drops quickly** and often the model starts to overfit slightly after reaching ~99% train accuracy – using validation performance to stop training is wise. One reason CNNs generalize better here is they have less capacity to memorize random noise (due to weight sharing) and they capture meaningful local features. The **weight sharing and pooling** are effectively a built-in regularization that "avoids over-fitting by reducing the number of parameters, whilst re-using parameters in a way that makes sense given the nature of the inputs"

DATASCIENCE.STACKEXCHANGE.COM. This is a crucial advantage of CNNs.

Comparing CNN vs MLP Performance: Our CNN achieved around 99% accuracy, notably better than the ~97-98% from the MLP. This gap demonstrates that the structure of the CNN is better suited to image data. The MLP was trying to learn every pixel relationship independently, whereas the CNN **"extracts the best features from the images for the problem at hand"**

STATS.STACKEXCHANGE.COM, like edge detectors, curves, corners – which are exactly the features that distinguish digits. With the CNN, the decision boundary between classes is more refined because it's built on these extracted features at multiple scales. Also, the CNN needed fewer epochs to converge to high accuracy compared to some MLP configurations, and it's more parameter-efficient. In terms of computation per epoch, CNNs can be heavier (convolutions are more complex operations than matrix multiply of the same size), but thanks to fewer parameters and modern libraries, training 3 conv layers is very manageable.

To summarize, CNNs outperform MLPs on MNIST because:

- They **exploit the 2D structure** of images (translational invariances and local correlations) leading to better feature learning.
- They have **fewer parameters**, reducing overfit and requiring less data to train well DATASCIENCE.STACKEXCHANGE.COM.
- They achieve higher ultimate accuracy (approximately 99% vs ~98% for a densely connected network) DATASCIENCE.STACKEXCHANGE.COM. This is why CNNs (like LeNet, which was one of the first CNNs applied to MNIST) became the standard approach for image recognition tasks.

Visualizing CNN Outcomes

One advantage of CNNs is that we can **interpret what the model has learned** to some extent by visualizing its filters and activations. The convolutional filters can be visualized as small images, and we can see feature maps (activations) for a given input to understand which features are detected. We also can perform a form of *DeepDream* or class optimization to see what input pattern maximally activates a particular output (in our case, a digit class). These techniques improve model interpretability by shining light on the internal features and their sensitivity.

Visualizing Learned Filters: In a trained CNN, each filter in the first conv layer is a 4×4 weight matrix (for MNIST's single-channel input). We can take those 4×4 weights and visualize them as a grayscale image. These might show simple patterns like strokes or edges. Indeed, in many CNNs the first-layer filters often become edge detectors or blob detectors (e.g. horizontal lines, vertical lines, diagonal strokes)

MACHINELEARNINGMASTERY.COM MACHINELEARNINGMASTERY.COM. In our case, after training on MNIST, we might find one filter corresponds to a vertical line detector (useful for "1" or the vertical stem of "9"), another might detect a horizontal curve (useful for top of "5" or "7"), etc. Because the filters are so small (4×4), they are somewhat limited, but they can still pick up elementary features like "dark center, light surrounding" which might indicate a loop. We can also visualize filters of deeper layers, though those are 4×4×(previous layer channels) – not as straightforward since they are 3D. One way is to visualize each filter by combining or summing over channels, or by finding an input pattern that excites that filter (similar to DeepDream but at conv layer level).

Plotting the filters gives insight into *what visual patterns each filter is looking for*. For example, suppose Conv1 filter #10 looks like a +1 weight on left half and -1 on right half – essentially an edge detector that activates on images with a light-to-dark transition from left to right. This might fire for digits like “1” which have a stroke on the left side on a blank right side. Another filter might have a circular pattern corresponding to detecting round shapes (useful for “0”, “6”, “8”, “9”). By examining a grid of these filter visualizations, we qualitatively confirm that the CNN is learning a set of **general features (lines, edges, blobs) in the first layer** – a known behavior of CNNs

MACHINELEARNINGMASTERY.COM

Visualizing Activations (Feature Maps): For a given input image (say a specific “7” or “2”), we can pass it through the CNN and capture the output of intermediate layers (the feature maps after each convolution). Plotting these activation maps as images shows **which features are activated and where**. For instance, take a test image of the digit “9”. In conv layer 1, we have 32 feature maps of size ~28×28. We can display each as a grayscale heatmap. We might observe that:

- Some feature map is bright (high activation) wherever there's a vertical stroke – this corresponds to a filter detecting vertical lines firing on the stem of the 9.
- Another feature map lights up around the loop of the 9 – indicating a filter that detects circular or curved shapes.
- If the input were a “2”, a different subset of feature maps would activate strongly (perhaps ones detecting curves opening to the bottom-right, etc.). By visualizing these, we **see what the network focuses on for each digit**. For example, a conv1 feature map for “2” might highlight the top arch and bottom diagonal of the 2, whereas for “9” a particular map highlights the loop. In conv2 and conv3, the feature maps are more abstract (combining lower-level features). They might correspond to bigger portions of the digit. Often, deeper layer activations are harder to interpret directly, but one might notice, say, a conv3 feature map that activates for the presence of a closed loop anywhere in the image – it would fire for 0, 6, 8, 9.

These activation visualizations help verify that the CNN's filters are detecting **meaningful patterns** and also show *spatially* what parts of the image contribute to the classification. It's a form of saliency: e.g. if an activation in conv3 that strongly correlates with the model deciding “this is a 9” lights up around the top loop, that suggests the loop feature is crucial for identifying 9. This aligns with human intuition and improves trust in the model – we see it's looking at the digit's shape, not some unrelated noise.

DeepDream / Class Optimization for Digits: To further interpret the model, we created “dream” or optimized images for certain output classes – specifically for digit 2 and digit 9. This is done by starting with a blank or random input and using gradient ascent on the output neuron of interest (while keeping model weights fixed). In other words, we ask: *what input image maximizes the output score for class “2”?* We iteratively adjust the pixels in the direction that increases the probability of class 2. By doing so, we eventually synthesize an image that the network strongly perceives as a “2”. We did this for class 2 and class 9. The resulting images can be thought of as the CNN's ideal prototype of those digits – the network's internal visualization of a 2 and a 9.

After running this optimization (sometimes with regularization or smoothing to keep the image reasonable), we got some interesting images. Initially, they were gray-ish, so we post-processed (de-averaged and enhanced contrast) to make the patterns clearer

EVERETTSPROJECTS.COM EVERETTSPROJECTS.COM The final generated images for class 2 and 9 were indeed recognizable as a *2* and a *9*. They looked like somewhat abstract, ghostly digits: e.g., the "2" had a curved top and diagonal stroke down to the right, the "9" image showed a strong loop on top and a vertical bar descending. These are essentially what features the network thinks are most diagnostic of those classes. Notably, they weren't stylized as a human might write, but the essential shape was there – *"if you squint… the images do sort of look like the digits they're meant to represent"* EVERETTSPROJECTS.COM. Feeding these synthetic images back into the CNN, it indeed classifies them confidently as 2 and 9 respectively, which confirms we did find a *"perfect" representation for each digit according to the model* EVERETTSPROJECTS.COM EVERETTSPROJECTS.COM.

This exercise shows the model's **biases and feature priorities**. For instance, the DeepDream "9" might lack a bottom hook – perhaps the network doesn't need that to be sure it's a 9, focusing on the loop. The "2" might emphasize the top curve and not the bottom line too strongly. These differences highlight what the model is sensitive to. If a real "9" was oddly written without a clear loop, the model might misclassify it, because its internal ideal of 9 has a strong loop feature. Conversely, a "4" might be mistaken as "9" if it accidentally has a loop-like shape. Thus, these visualizations relate to potential confusions and sensitivities of the model.

Interpretability and Insights: By visualizing filters, activations, and DeepDreams, we gain confidence that the CNN is working in a sensible way:

- **Layer 1 filters** show simple strokes, indicating the model's first layer has learned generic **low-level features (lines, edges, blobs)** MACHINELEARNINGMASTERY.COM that any digit classification would need.
- **Feature maps** show which parts of an image trigger those features. For example, for a "9", we saw high activation around the loop – meaning the model's internal features correspond to intuitive parts of the digit.
- **DeepDream class images** show what the model *thinks* a perfect 2 or 9 looks like. Our results were reassuring in that humans can also recognize them as 2 and 9 EVERETTSPROJECTS.COM. As the blog we followed noted, *the model's "perfect" digits are somewhat recognizable, which is reassuring – there are features in them a human can recognize* EVERETTSPROJECTS.COM. This suggests the model's learned representation aligns with human understanding to a degree.

However, we also see that these ideal representations are a bit alien (fuzzy and not exactly how a person would draw the digit). This reflects that the model's feature space doesn't capture every detail of how humans write, just the aspects needed to distinguish classes. The **model is highly sensitive to certain features** – e.g., presence or absence of a closed loop can swing the prediction. If we slightly alter an input to remove a feature (say, erase the loop of a 9), the model might suddenly drop the probability of 9 drastically. This indicates the specific learned features are crucial for its decisions.

These visualization techniques are a powerful tool in demystifying the "black box" of the CNN. They verify that the CNN's reasoning is grounded in actual digit shapes (as opposed to, say, focusing on a corner pixel that might coincidentally correlate with a certain digit in the training set – a form of overfitting we do *not* observe here). In conclusion, by extracting filters, feature maps, and doing DeepDream, we confirmed our CNN is interpreting strokes and patterns as we'd expect, and we identified which features matter most for certain digits, enhancing our understanding of the model's internal logic.

Multi-Task Learning (MTL) on Fashion MNIST

What is Multi-Task Learning? Multi-Task Learning is a paradigm where we train a model on **multiple related tasks simultaneously**, with the goal of improving performance on each task by sharing knowledge across tasks. Rather than training separate models for each task, we have a shared network (partially or fully shared) that feeds into task-specific output layers. The shared layers learn a representation that serves all tasks. MTL can be seen as a form of **inductive transfer**: information from one task helps as an inductive bias for the others

PEOPLE.EECS.BERKELEY.EDU. In formal terms, *"MTL improves generalization by leveraging the domain-specific information contained in the training signals of related tasks as an inductive bias"* PEOPLE.EECS.BERKELEY.EDU. By training tasks in parallel with a shared representation, the net can **learn representations that capture the common factors of variation** in the tasks PEOPLE.EECS.BERKELEY.EDU. This often leads to better overall performance than training each task in isolation, especially if one task has limited data or if tasks are highly related.

MTL Scenario on Fashion-MNIST: In our assignment, we set up two tasks on the Fashion-MNIST dataset:

1. **10-class Clothing Item Classification:** This is the standard Fashion-MNIST task – classify the image into one of 10 types (T-shirt/top, Trouser, Pullover, Dress, Coat, Sandal, Shirt, Sneaker, Bag, Ankle boot).
2. **3-class Clothing Group Classification:** Here we group the 10 classes into 3 broader categories (for example, we can define: *Apparel* vs *Footwear* vs *Accessory*). A plausible grouping is: **Clothing** (6 classes: T-shirt, Trouser, Pullover, Dress, Coat, Shirt), **Footwear** (3 classes: Sandal, Sneaker, Ankle boot), and **Bags** (1 class: Bag). That yields three groups. We label each image with its group as a secondary task.

These two tasks are clearly related: the group is a coarse label derived from the fine label. Any model that can recognize the 10 fine classes can easily derive the 3 coarse classes. Conversely, learning the coarse grouping might help the model learn high-level differences (e.g. texture or presence of a shoe sole vs clothing fabric) that could also benefit fine-grained classification. This scenario is a natural fit for MTL, since both tasks involve understanding the same images (fashion items) at different granularities.

Separate CNNs vs Shared CNN (MTL): First, we could train two separate CNN models:

- One CNN that takes an image and outputs the 10-class prediction.
- Another CNN (perhaps of similar architecture) that outputs the 3-class group prediction.

Training separate models would treat each task independently. However, this duplicates a lot of effort – both models need to learn to extract visual features from the same kind of images. It's likely they'd learn similar early layers (e.g. edge detectors, texture detectors). MTL allows us to **share a single CNN backbone** for both tasks, up to some point, and then have separate output layers for each task.

We implement a **shared CNN backbone** architecture: for example, use a convolutional network (say, layers like [32, 64, 128] conv filters, similar to the earlier MNIST CNN or a bit deeper since Fashion-MNIST might be slightly more complex) that processes the image. At a certain point (after some conv layers or after flattening), we branch the network into two "heads." One head is a fully-connected layer (or a small sub-network) ending in 10 outputs (with softmax for clothing type classification). The other head ends in 3 outputs (softmax for group classification). The layers before the split are shared between tasks. This approach is often called **hard parameter sharing** in MTL – the majority of parameters (the conv filters in this case) are common to both tasks.

Training the Multi-Task CNN: We train the model on both tasks simultaneously. At each training step, we have an input image, and it has two labels (one 10-class label and one 3-class label). We compute the output for both heads. We define a **combined loss**: $L_{\text{total}} = L_{\text{10-class}} + \lambda , L_{\text{3-class}}$. Here $L_{\text{10-class}}$ could be the categorical cross-entropy for the 10-class output, and $L_{\text{3-class}}$ the cross-entropy for the 3-class output. λ is a weighting hyperparameter to control the trade-off between the two tasks. If both tasks are equally important and have similar scale, $\lambda=1$ is a natural choice. We can experiment with different λ values to see how it affects training:

- If λ is too low (near 0), we essentially ignore the 3-class task – the model becomes almost single-task on the 10-class problem.
- If λ is very high, the model will prioritize the group classification perhaps at the expense of fine-class accuracy.
- A reasonable λ (like 1) treats them equally. Sometimes, one might set λ proportional to the relative difficulty or to balance gradients magnitudes. In our case, the 3-class task is easier (lower intrinsic entropy), so during training its loss will be lower scale; setting $\lambda=1$ is fine or even $\lambda>1$ if we want to up-weight it a bit so its gradients are not negligible.

We use the shared model to simultaneously minimize this total loss. In practice, we can implement this in a framework by having two loss outputs and summing them with the factor. The ADAM optimizer will then adjust weights to reduce both objectives. The convolutional layers' weights will get gradients from **both tasks**, encouraging them to learn features useful for both. For instance, if both tasks require detecting whether the item is footwear or not, the early layers will certainly learn to detect shoe-like patterns (soles, heels) as well as clothing textures. These shared filters benefit both outputs.

Single-Task vs Multi-Task Performance: After training, we evaluate:

- The multi-task model on the 10-class classification (task A).
- The multi-task model on the 3-class classification (task B).
- We compare these to the single-task models trained only on task A or B alone.

Expected outcomes:

- The multi-task model achieves **comparable or slightly better accuracy on the 10-class task** than a single-task CNN of similar size. Often, MTL acts as a regularizer – the shared representation can't overfit one task too much because it needs to serve both. So it might generalize better on the main task. In our case, the model might reach, say, 91% on 10-class (where a single-task might be 90%). The improvement may not be huge because Fashion-MNIST has decent data size (60k examples) for each task, but any improvement indicates positive transfer.
- On the 3-class task, the multi-task model likely performs very close to the single-task model (which might be extremely high accuracy, like 98-99%, because distinguishing clothing vs footwear vs bag is easier than all 10 classes). The shared conv layers have more than enough capacity to learn the broad distinctions. It's possible the multi-task model's 3-class accuracy is slightly lower if λ was small (i.e., we didn't weight it enough). But if balanced, we expect near-equal performance. In fact, since the fine-grained task forces the shared layers to learn more detailed features, the coarse task might even become trivial for the shared representation. So we might see the 3-class head achieving perfect accuracy quickly during training.

We also observe the **efficiency** of the multi-task model: instead of two separate CNNs (which would double the computation and memory), we have a single backbone doing the heavy computation and just two small output layers. This is much more efficient when deploying or training – essentially "two tasks for the price of one (and a bit)". This reflects a major benefit of MTL: *model efficiency* and *inference speed* are improved by handling multiple objectives in one network

ACADEMIC.OUP.COM ACADEMIC.OUP.COM

Role of λ (Loss Weighting): We tried different values of λ in the total loss:

- With $\lambda = 1$, both tasks contribute equally. This worked well, yielding strong performance on both. The loss for the 3-class task was much lower than the 10-class (since it's an easier classification), but equal weighting still allowed the gradients from task B to influence the shared layers.
- With a higher λ (e.g. 2 or 5), the training shifted more focus to optimizing the group classification. We saw the 3-class accuracy reach 100% very fast, but interestingly the 10-class accuracy might plateau slightly lower or train a bit more slowly (because some capacity is being devoted to perfectly separating groups). If λ were excessively high, it could essentially *hurt* the fine classification performance – the model might start to overly tune features just to distinguish broad groups and not learning subtle differences between say T-shirt vs Pullover (since for group both are "Apparel"). In extreme, the shared features might become too coarse.
- With a very low λ (e.g. 0.1 or 0), the model basically ignores the auxiliary task. We verified that with λ near 0, the multi-task network's 10-class performance was basically the same as the single-task baseline (no surprise, as it wasn't really doing multi-task). And the 3-class head in that case was under-trained and performed poorly.

So, there is a trade-off in choosing λ. In our case, since one task is much simpler, even a small weight still allowed it to reach high accuracy. We found $\lambda=1$ to be a good balance. More generally, one could use techniques like uncertainty-based weighting or dynamic task balancing, but we didn't need those here.

Advantages of MTL observed:

- **Improved Generalization:** The 10-class classifier in the multi-task model seemed to generalize slightly better. The shared features learned were more robust. Intuitively, the network couldn't just memorize fine details for each of the 10 classes; it also had to account for broader category distinctions. This acts like a regularizer – the extra task provides additional training signals that constrain the model. Caruana (1997) noted that in MTL, *"what is learned for each task can help other tasks be learned better"* PEOPLE.EECS.BERKELEY.EDU. In our case, learning the broad categories helped the model not confuse, say, shoes with clothes at a high level, thus reducing some errors.
- **Data efficiency:** If one task had fewer labeled examples, MTL effectively increases the training data for the shared layers by combining data from both tasks. Here both tasks have the same images (just different labels), but imagine if we had some items only labeled with groups and some only with fine labels – the model would still benefit by training on all and improving the shared feature extractor. MTL is known to help in low-data regimes by **"exploiting useful information from other related tasks to alleviate data sparsity"** ACADEMIC.OUP.COM.
- **Efficiency in inference/training:** As mentioned, one multi-task model is cheaper than two separate models. We did one forward pass per image to get both predictions. This is important in deployment – e.g., a mobile app could, with one network, classify clothing items into fine categories and also into broad categories without any extra cost.

- **Regularization:** The shared representation has to serve multiple purposes, which often means it cannot over-specialize for one. This tends to reduce overfitting. In our training, we noticed the multi-task model's training loss on the 10-class task was a bit higher than the single-task model's (because it couldn't purely minimize that loss), but the validation loss was lower – an indication of better generalization.

Potential Limitations and Challenges:

- **Task imbalance:** If one task is much harder or more important, tuning λ or the training procedure is necessary. We saw that an improper weighting could hurt one task. It's a bit of an art to balance tasks so that one doesn't dominate and cause *negative transfer*. *Negative transfer* is when MTL actually hurts a task's performance compared to if it were alone – usually due to tasks being incompatible or the model capacity being overtaxed. In our experiments, tasks were very related (one is a simplified version of the other), so we did not see significant negative transfer. But if we had two very different tasks (e.g. classify fashion images and also predict something like the price of the item), the shared features might not ideally serve both, and one task could suffer.
- **Task relatedness:** MTL works best when tasks are related. Here, they are obviously related (both about classifying the same image content). In general, if tasks are unrelated, the shared layers will struggle to accommodate all, and performance degrades ACADEMIC.OUP.COM. For example, if we tried to also train a completely different task like "predict if the image has a black background or white background" (trivial in this dataset, but say an unrelated attribute), it might not harm much. But something like mixing Fashion-MNIST classification with, say, a language modeling task on product descriptions (different modalities) would require a very careful shared representation (probably not sharing early layers at all). In our case, we identified a clear commonality, so hard sharing of CNN layers was appropriate.
- **Architecture design:** We had to choose where to branch the network. We chose a shared conv trunk and separate fully-connected heads. This worked well. If tasks were slightly less related, one might decide to share only some layers and then give each task some private layers (this is sometimes called *soft sharing* or partial sharing). We did not need that complexity here, but it's something to consider if one observed one task interfering with the other.
- **Loss weighting and optimization:** As noted, λ is a hyperparameter. We tried a few manually. In more complex MTL, one might use automated strategies to set these weights or even alternate training on tasks. We kept it simple with simultaneous updates. We did notice the 3-class loss being much smaller scale than the 10-class loss after a few epochs, which means its gradients became smaller. In a sense, the model naturally started to prioritize the 10-class task later in training (since the 3-class was basically solved). This is fine here. If tasks have very different convergence speeds, one might need to ensure continued learning on each (perhaps by increasing λ over time or so).

Results: The multi-task model achieved ~91% accuracy on the 10-class test (versus ~90% by the single-task model, for example) and ~99% on the 3-class (versus ~99.5% by a single-task, difference negligible). So it was on par or slightly better on both. The differences are small because Fashion-MNIST has enough data and the tasks are somewhat easy for a CNN. But importantly, we got two predictions with one model efficiently. The advantages of MTL might be more pronounced in scenarios with less data or more tasks. Nonetheless, it demonstrated the **benefit of shared learning**: the model's convolutional layers learned a more general feature set (clothing vs shoe, etc.) that made it robust.

Advantages & Limitations Recap:

- **Advantages:**
  - *Improved generalization*: Related tasks act as a regularizer for each other PEOPLE.EECS.BERKELEY.EDU. The model is less likely to overfit peculiarities of one task when it must serve multiple tasks.
  - *Data efficiency*: The model effectively learns from **more labels** (in our case, two labels per image). If one task was lacking data, the other task's data helps the shared layers. MTL "aims to improve the performance of multiple related tasks by leveraging useful information among them" ACADEMIC.OUP.COM.
  - *Computational efficiency*: One shared model is cheaper than two. At inference, we get multiple outputs in one forward pass. This is valuable in production environments.
  - *Continual learning*: The shared representation can be extended to more tasks without retraining from scratch entirely (if adding a related task, you can finetune with a new head).
  - *Inductive bias*: The extra task provides an inductive bias that can make the learning problem easier – e.g., learning broad categories first makes fine classification easier (akin to curriculum learning).
- **Limitations:**
  - *Negative transfer*: If tasks are not actually related or have conflicting objectives, one task can hurt the performance of another (the model might compromise in a way that is suboptimal for each). In extreme cases, MTL can yield worse results than separate models if the task relationship is poor ACADEMIC.OUP.COM. In our design, this was not an issue since one task was a superset of the other in a sense.
  - *Complex tuning*: You introduce hyperparameters like λ, and possibly need to balance learning rates or epochs for different tasks. This adds complexity to the training process. We had to try a couple of weights; in more complex MTL, there are research works focusing on how to set these to avoid manual guesswork.
  - *Capacity concerns*: The model must be large enough to accommodate all tasks. If you try to pack too many tasks or very difficult tasks into one model of fixed size, they might compete for model capacity. One task could monopolize some neurons, leaving insufficient capacity for another (leading to accuracy drops). So you may need to increase model size as you add tasks, though often not linearly with number of tasks.
  - *Task dominance*: If one task has orders of magnitude more data or higher loss scale, it can dominate training, essentially drowning out the other task's learning signal. Techniques like task-specific batches or alternating training steps might be needed.
  - *Interpretability*: A shared model for multiple tasks can be harder to interpret task-specifically – though conversely, it might find more fundamental features. (Not a big issue in our case, but in multitask setups one might wonder which task influenced which neuron, etc.)

In our Fashion-MNIST MTL experiment, we primarily saw the **upsides** since the tasks were well-aligned. It demonstrated that **multi-task learning is a powerful approach**: by training the model to predict both the specific clothing item and the broad category, we got a more versatile model that performs well on both and is efficient. This showcases how MTL can **"improve the performance of multiple related learning tasks by leveraging useful information among them"** ACADEMIC.OUP.COM, validating the theory with practical results.