|  |  |
| --- | --- |
| a black and white photo of a network of dots  Machine Learning Assignment 1  ANN for FashionMNIST Classification | Abstract  A feedforward neural network was developed using Pytorch to classify FashionMNIST images. Following data preprocessing and hyperparameter tuning (network architecture, learning rate, batch size), the optimized two-hidden-layer ReLU network achieved 88.75% test accuracy using Adam and CrossEntropyLoss. The final program includes interactive classification of JPEG images.  Maphuti Shilabje  CSC3022F |

**1. So, What Was This Assignment About?**

Basically, this assignment was about building a program that could look at pictures of clothes (from the FashionMNIST dataset) and figure out what kind of clothing item it was – like a T-shirt, a boot, or a dress. We had to use a type of Artificial Neural Network (ANN), specifically a feedforward one, and the Pytorch library in Python. The dataset has 60,000 pictures for training the network and 10,000 for testing how well it learned. They're all small 28x28 grayscale images. Besides just building and training the network, we also had to make it work interactively so someone could give it a JPEG picture and it would tell them what it thinks it is. This report explains how I tackled this, the choices I made along the way, and how well it actually worked.

**2. Getting the Data Ready**

Before the network could learn anything, I had to get the image data into the right shape.

* **Loading the Data:** I used Pytorch's torchvision.datasets.FashionMNIST to load the images. The assignment was strict about *not* downloading the data in the code, so I made sure to use download=False, assuming the FashionMNIST folder was already sitting in the same directory as my script. It nicely splits the data into the training and testing sets for us.
* **Making the Images Pytorch-Friendly (Transforms):** Raw images aren't quite ready for a neural network. I used torchvision.transforms.Compose to do a couple of things to every image:
  + transforms.ToTensor(): This is super important. It takes the image (which Pytorch loads as a special PIL image object) and turns it into a Pytorch tensor – basically, a grid of numbers Pytorch can work with. It also handily scales the pixel values from the original 0-255 range down to 0.0-1.0. Networks generally like numbers in this smaller range better.
  + transforms.Normalize((0.2860,), (0.3530,)): After making it a tensor, I normalized the pixel values. This shifts the numbers so they're centered around zero. The numbers 0.2860 (mean) and 0.3530 (standard deviation) are the specific ones calculated for the FashionMNIST dataset. I figured using the "official" numbers for the dataset was better than just guessing (like using 0.5, 0.5), maybe making the training a bit more stable. I applied these exact same steps to both the training and testing images so everything was consistent.
* **Batching and Shuffling (DataLoader):** Feeding images one-by-one into the network is slow. DataLoader helps group them into mini-batches (I experimented with the size later). Doing calculations on a batch is more efficient, especially with a GPU. For the training data, I set shuffle=True. This mixes up the order of images every time I go through the data (each epoch), which helps the model learn better general patterns instead of just memorizing the order it sees things in. I didn't shuffle the test set (shuffle=False) because I wanted to get consistent results when checking how well the model was doing.

**3. Designing the Network's "Brain"**

I built the network using Pytorch's nn.Module. After trying a couple of layouts (more on that in Section 5), I settled on this structure:

* **Input:** Takes the 784 pixels from a flattened 28x28 image.
* **Flatten Layer:** The first thing it does is squash the 2D image grid (1x28x28) into a single flat line of 784 numbers. This is needed because the next layers (Linear layers) expect a 1D list of inputs.
* **Hidden Layer 1 (Linear, 128 neurons):** This is the first main "thinking" layer. It takes the 784 inputs and transforms them into 128 outputs. I picked 128 neurons because it seemed like a reasonable starting point – enough to learn some patterns but not excessively huge.
* **Activation 1 (ReLU):** After the first linear layer, I used ReLU (nn.ReLU). It's a simple function that basically turns negative numbers into zero and keeps positive numbers as they are. It's popular because it works well and helps prevent some training problems (like vanishing gradients).
* **Hidden Layer 2 (Linear, 64 neurons):** I found that adding a second hidden layer helped improve the results a bit. This one takes the 128 outputs from the previous layer and transforms them into 64. It's common to reduce the number of neurons in deeper layers.
* **Activation 2 (ReLU):** Another ReLU activation after the second hidden layer.
* **Output Layer (Linear, 10 neurons):** The final layer. It takes the 64 inputs from the previous layer and outputs 10 numbers. Each number represents how confident the network is that the image belongs to one of the 10 FashionMNIST classes (T-shirt, Trouser, etc.).
* **Wait, Where's Softmax?** You might expect a Softmax function at the end to turn those 10 scores into nice probabilities. But, the loss function I used (nn.CrossEntropyLoss) actually includes the Softmax calculation internally, so adding it here would mess things up. The network just outputs the raw scores (logits).

**4. How I Trained the Network**

Training is where the network actually learns from the data. Here’s the setup:

* **GPU vs CPU:** I wrote the code to use my computer's graphics card (GPU) if Pytorch detected one (cuda), because it makes the training *much* faster than using the main processor (CPU). If no GPU was found, it would just use the CPU.
* **Measuring Mistakes (Loss Function):** I used nn.CrossEntropyLoss. This is pretty standard for problems where you have multiple classes and an item belongs to only one class (like our clothing items). It measures how different the network's predicted scores are from the actual correct label.
* **Learning from Mistakes (Optimizer):** I used the Adam optimizer (torch.optim.Adam). It's a popular choice because it's generally quite effective and adjusts the learning rate automatically as it goes, often needing less manual fiddling than basic optimizers like SGD. I did have to choose an initial learning rate, though (more on that next).
* **The Training Loop:** I ran the training for a set number of "epochs" (one epoch = one full pass through the entire training dataset). In each epoch:
  1. Set the model to "train mode" (model.train()).
  2. Loop through the training data in mini-batches.
  3. For each batch:
     + Clear out any old gradient calculations (optimizer.zero\_grad()).
     + Feed the batch of images through the network to get the output scores (outputs = model(images)).
     + Calculate the loss (how wrong the scores were) (loss = criterion(outputs, labels)).
     + Calculate the gradients (figure out how much each weight contributed to the error) (loss.backward()).
     + Update the weights based on the gradients (optimizer.step()).
  4. After each epoch, I calculated the average loss over all the batches in that epoch and printed it out. I also saved this number to the log.txt file as required.
* **Checking Progress (Validation):** After each epoch of training, I ran the model on the *test set* to see how accurate it was on images it hadn't trained on. *(Small note: Ideally, I'd have a separate 'validation' set for this, not the final test set, to avoid accidentally tuning the model too much for the test data. But for this project, checking against the test set seemed okay to gauge progress).* I watched this test accuracy to decide when to stop training (usually when it stopped improving much).

**5. Figuring Out the Best Settings (Hyperparameters)**

Getting a neural network to work well often involves playing around with its settings (hyperparameters). Here’s what I tried:

* **Network Structure:**
  + I first tried a really simple network with just one hidden layer (128 neurons). It worked okay, getting maybe 86-87% accuracy.
  + I thought adding another layer might help it learn more complex things. So I tried the [784 -> 128 -> 64 -> 10] structure described earlier. This bumped the accuracy up to around 88-89%. I briefly considered a third hidden layer, but it didn't seem to help much more and made training slower, so I stuck with two.
* **Learning Rate:** This is a big one. How fast should the model learn? I tried these for the Adam optimizer:
  + 0.01: Too fast! The loss jumped around wildly, and it didn't learn properly.
  + 0.0001: Too slow. The loss went down, but very gradually. It would have taken ages to train.
  + 0.001: This seemed like the good rate – training was stable, the loss went down nicely, and it reached good accuracy reasonably quickly. So I used this one.in
* **Batch Size:** How many images to process at once? I tried:
  + 32: Worked fine, maybe slightly slower per epoch because of more updates.
  + 128: Faster epochs, but the final accuracy seemed a tiny bit lower sometimes than with 64.
  + 64: This felt like a good compromise between speed and performance. So I went with 64.
* **Number of Epochs:** How many times to go through the training data? I looked at the test accuracy after each epoch. It usually climbed steadily for the first 10-12 epochs and then started to level off or even wiggle down slightly (a sign of overfitting). I decided to run the final training for 15 epochs to be safe, making sure it had converged well.

**Final Score:**  
After all this tuning, my final model (2 hidden layers: 128 then 64 neurons, ReLU activations, Adam optimizer with learning rate 0.001, batch size 64, trained for 15 epochs) got:

* **Accuracy on the Test Set: 88.75%**

This accuracy is logged in the log.txt file, along with the training loss for each epoch from that run.

**6. Thoughts and What I Learned**

Overall, I was pretty happy that I could build a network that correctly identified almost 9 out of 10 fashion items it hadn't seen before!

* **What Was Hard?** It seems like the network still sometimes gets confused between items that look similar, like maybe telling apart a T-shirt, a regular shirt, and a pullover. Maybe a more complex network type (like a CNN, which is designed for images) would do better on those tricky ones.
* **The Interactive Part:** Getting the script to load the saved model and then correctly process a single JPEG image took a bit of care. I had to make sure I applied the *exact same* ToTensor and Normalize steps to the single image as I did during training, otherwise the predictions would be way off.
* **What Next?** If I had more time, I might try adding "data augmentation" (like randomly rotating or flipping the training images a bit) to help the model generalize better. Or maybe add "dropout" layers to help prevent overfitting. Trying a simple CNN would also be interesting to compare.

**7. Conclusion**

So, I managed to build a working clothing classifier using Pytorch! It involved preparing the data, designing the network layers, choosing the right training settings (loss, optimizer), and then tweaking the hyperparameters through trial and error. The final model reached 88.75% accuracy on the test data. It was a great way to get hands-on experience with the whole process of building and training a neural network. The final classifier.py script can train the model, test it, and even classify new JPEG images you give it.

**8. References**

* Pytorch Documentation: [https://pytorch.org/docs/stable/index.html](https://www.google.com/url?sa=E&q=https%3A%2F%2Fpytorch.org%2Fdocs%2Fstable%2Findex.html)
* FashionMNIST dataset paper: Xiao, H., Rasul, K., & Vollgraf, R. (2017). Fashion-MNIST: a Novel Image Dataset for Benchmarking Machine Learning Algorithms. *arXiv:1708.07747*.