USE CNN TO DISTINGUISH IMAGES WITH CPU AND GPU PERFORMANCE ANALYSIS

EDRIA LI

Applied Mathematics Department, University of Washington, Seattle, WA
yijueli@outlook.com

Abstract. We successfully trained a Fully Connected Deep Neural Networks (FCNs) with the constraint of incorporating up to 100K weights to distinguish images in the FashionMNIST dataset with test accuracy of 88.5156%. FCN models with 50K and 200K weight constraints were also examined. We further implemented and trained a Convoluted Neural Networks (CNNs) with constraints of 100K weights model and achieved 90.7031% test accuracy. Different sizes of CNN weights with parametrized convolutional, pooling and FC layers were experimented. CPU and GPU training times were compared for all models. Feature maps of the convolutional layers were visualized.

1. Introduction and Overview

We are provided with FashionMNIST data with 60000 samples in training and 10000 samples in test sets. The training set is further split to training (90%) and validation (10%) where each image is represented in pixels of size 28×28.

Our task is to design a Fully Connected Deep Neural Network (FCN) with 100K weights contraints to classify the dataset with test accuracy above 88%; and to design a Convolutional Deep Neural Network (CNN) model with convolutional, pooling, and FC layers with up to 100K weights and to explore test accuracy. We used our FCN model generated in our previous report as our baseline configuration for current tasks. We generate models with an adjustable number of hidden layers and neurons in each layer with ReLU activation. Cross Entropy Loss function were used for training. Adam optimizers are used in both FCN model and FC layer in our CNN model due to it's observed robustness in our previous experiments. Similarly, for training purpose, data were passed in through batches with batch size 512. We compared the test accuracy with our FCN and CNN model variants and discussed the training times in both CPU-only and CUDA-enabled GPU scenario and efficiency inherited in each variant. Finally, we visualized the feature map of several input samples of the two convolutional layers respectively.

2. Theoretical Background

2.1. **Deep Neural Network Basics.** Neural Networks have their names loosely inspired by neurobiology with similar structure modeling neurons [1]. One key design consideration for neural networks is the architecture, which refers to the overall structure of the network. The neural networks are organized into groups of units called layers, which are further arranged in a chain structure, where the first layer is $\mathbf{y}^1 = f^1(\mathbf{W}^{1\top}\mathbf{x}^1 + \mathbf{b}^1)$, and the second layer $\mathbf{y}^2 = f^2(\mathbf{W}^{2\top}\mathbf{y}^1 + \mathbf{b}^2)$ and so on, where weights $\mathbf{W}^\top\mathbf{x}$ can be elaborated itemwise in $\sum_{i=1}^{n} x_i w_i$. For forward propagation, the activation function as denoted above f can be represented in many forms, such as(1) Sigmoid function: $\sigma = f = \frac{1}{1+e^{-x}}$; (2) tanh function: $f = \tanh(x)$; (3) Rectified Linear Unit function (ReLU): $f = \max(x, 0)$; or (4) Leaky parametric ReLU: $f = \max(x, ax)$. ReLU is a default activation function recommended for use with most feedforward neural networks [1].

2.2. **Convolutional Neural Networks.** Convolutional neural networks (CNNs) are specialized neural network for processing data that has a known, grid-like topology (such as image data which are essentially 2D grid of pixels) and use convolution in place of general matrix multiplication in at least one of the layers [1]. Mathematically, convolution is an operation on two functions, i.e. $s(t) = (x * w)(t) = \int x(a)w(t-a)\mathrm{d}a$, where x is referred as **input** and function w as **kernel** [1]. Its discrete form is: $s(t) = \sum_{a=-\infty}^{\infty} x(a)w(t-a)$. For a 2D image as input, we will need a 2-dimensional kernel for our 2D convolution operation: $S(i,j) = (K * I)(i,j) = \sum_m \sum_n I(i-m, j-n)K(m,n)$. Discrete convolution, however, can be viewed as multiplication by a matrix, where different from traditional fully-connected layers, CNNs have sparse weights due to the fact that kernel is made smaller than the input. Thus gives us the motivation for computational benefits that we can shrink runtime order for m input and m output from $O(m \times n)$ to a sparsely connected network runtime order of $O(k \times n)$, where k denotes new output size after we define kernel and stride size. A typical layer of CNN consists of 3 stages: (1) performs convolutions and linear activation (such as affine transform); (2) nonlinear activation function, such as ReLU; (3) pooling function to modify the output layer.

2.3. **Pooling for CNNs.** A pooling function replaces the output with a summary statistic of the nearby outputs. Max pooling reports the maximum output within a rectangular neighborhood; other pooling functions include the average of a rectangular neighborhood, L^2 norm of rectangular neighborhood, or a weighted average (such as Gaussian normal filter) [1].

2.4. **Cross Entropy Loss and Gradient Descent.** Cross entropy is defined as $H(P,Q) = \mathbb{E}_{x\sim P}[-\log Q(x)]$ with P denotes distribution given and Q denotes distribution we trained. Loss function is defined as $\mathcal{L}(\hat{y}, y) = -(y \log \hat{y} + (1-y)\log(1-\hat{y}))$, where $\hat{y}$ denotes prediction and y is given. The cost function (or cross entropy loss) is defined as: $\mathcal{J}^m(y, \hat{y}; x) = \frac{1}{m}\sum_{i=1}^m \mathcal{L}^i(\hat{y}, y)$, where index i traverse through each sample and m is the total number of samples. Gradient descent is than used as an iterative approach to optimize the given function $\mathcal{J}$. The weights and bias can be updated as $\mathbf{W}_{k+1} = \mathbf{W}_k - \alpha\nabla_{\mathbf{W}}\mathcal{J}(\mathbf{W}_k; \mathbf{b})$ and $\mathbf{b}_{k+1} = \mathbf{b}_k - \alpha\frac{\partial}{\partial \mathbf{b}}\mathcal{J}(\mathbf{W}; \mathbf{b}_k)$ respectively, where α denotes learning rate.

2.5. **Regularization in FCNs.** Regularization terms can be further added to the cost function $\mathcal{J}$ as the follows: $\mathcal{J}^m(\mathbf{w}, b) = \frac{1}{m}\sum_{i=1}^m \mathcal{L}^i(\hat{y}, y) + \frac{\lambda}{2m}\sum_{i=1}^L \|\mathbf{w}^l\|_F^2$ where $\|\mathbf{w}^l\|_F^2 = \sum_{i=1}^l \sum_{j=1}^l (\mathbf{w}_{ij}^l)^2$, i.e. the squared sum of all weight terms.

2.6. **Initialization in FCNs.** Initialization strategies are used to "break symmetry" between different layers [1]. Xavier Normal uses distribution sampled from $\mathcal{N}(0, \sigma^2)$ where σ = gain $\times\sqrt{\frac{2}{\texttt{fan\_in}+\texttt{fan\_out}}}$, where `fan_in` preserves the magnitude of the variance of the weights in the forward propagation whereas `fan_out` does so in the backward propagation [2].

2.7. **Batch Normalization.** Batch normalization is an adaptive reparametrization method used especially in very deep models [1]. To normalize $\mathbf{H}$, a minibatch of activations, we replace it with $\mathbf{H}' = \frac{\mathbf{H}-\mu}{\sigma}$. During training time $\mu = \frac{1}{m}\sum_i \mathbf{H}_i$ and $\sigma = \sqrt{\delta + \frac{1}{m}\sum_i (\mathbf{H}-\mu)_i^2}$. We backwardly propagate through these operations for computing μ and σ, then apply them to normalize $\mathbf{H}$ [1].

3. Algorithm Implementation and Development

we used `Python` in which we import `torch.utils.data` for data loading; `numpy`, `torch`, and `torchvision` for computing; `sklearn.model_selection` for data split. We also import `matplotlib` and `seaborn` for plotting.

As we have elaborated our FCN Algorithm and Train/Test call function in the last report, here we present the detailed algorithm for CNN Model (Algorithm 1):

Algorithm 1 Setup CNN Parametrized Model with Convolutional, Pooling, and FC Layers

Setup `Class` using `nn.Module`.
Initialize model layers, pass in dataset convolution layer channels (`conv_channels`), number of kernel for each layer (`conv_kernel`), number of maximum-pooling kernel (`pool`), FC layer dimensions(`fcn_dims`), output dimension (`output_dim`), dropout rate for FC layer (`dropout`) and batch normalization switch (`batch_norm`) as parameters.
for i in range(length of `conv_channels` -1) **do**
    append the new convolutional layer by `nn.Conv2d(conv_channels[i] and [i+1])`
end for
Append the first FC layer by `nn.LazyLinear` to avoid dimension error
for i in range(length of `fcn_dims` -1) **do**
    append the new FC layer by `append(nn.Linear(fcn_dims[i], fcn_dims[i+1]))`
end for
Append the output layer
Load pooling, dropout, and batch normalization parameter values into the model.
Define model Forward Propagation `forward`
for each layer in `self.layers[:-1]` **do**
    if layer `isinstance` of `nn.Conv2d` **then** apply next convolution layer;
        if `batch_norm == True` **then** apply `nn.BatchNorm2d`;
        end if
        Activate inputs using `ReLU`
        if `pool` $\neq$ `None` **then** apply `nn.MaxPool2d` to layer output.
        end if
    end if
    if layer `isinstance` of `nn.Linear` or `nn.LazyLinear` **then** reshape input by `nn.Flatten()`
        Apply next fully connected layer;
        if `batch_norm == True` **then** apply `nn.BatchNorm1d` to layer output;
        end if
        Activate inputs using `ReLU`
        if `dropout` $\neq$ `None` **then** apply `nn.Dropout` to current layer.
        end if
    end if
end for

4. Computational Results

Computational resources are crucial in neural network model training, as most large language models or deep neural networks are trained using GPUs or TPUs with numerous petaFLOPs/s day. This give us the motivation to consider mainly in this report regarding computational budget of weights (including bias as parameters as well) that we can incorporate into FCN and CNN models. We consider FCN 100K, 50K, and 200K weights, and similarly 10K, 20K, 50K and 100K weights for CNN. The training for each model is timed for executions in either CPU only or CUDA-enabled GPU, for the purpose of runtime comparison for different computational resources and discussions on efficiency and necessity for certain model architectures.

4.1. **FCN 100K Configurations.** Due to the large input dimension of 28×28, in order to limit the weights within 100K budget, our first layer dimension is set to 100. We used Python command `summary` to know the exact size of our model. Based on our training experiences, we used Adam optimizer with its default learning rate 0.001 and experimented with Xavier Normal, and achieved

test classification accuracy of 88.5156% [1] (compared with 87.8223% baseline without initialization) as shown in the following Table 1 (see also Figure 1a and 1b in Appendix A for train loss curves and validation accuracy curves). "Mult.Adds" in the Table means the total number of operations that *multiply* weights $\mathbf{W}$ with inputs $\mathbf{x}$ and *add* the bias b (essentially $\mathbf{W}^\top \cdot \mathbf{x} + b$). As expected, for the small amount of operations (50.43 MB) it shows an increase in runtime for CUDA-enabled scenario due to the extra time used for memory allocation for CUDA device in the cache.

Init.	Weights	Test Acc.	CPU-only	with GPU	Mult.Adds	Low.Loss
Xavier	98,490	88.5156%	2m 32.93s	2m42.66s	50.43 MB	0.11, 17^{th} Epoch
Baseline	98,490	87.8223%	2m 30.35s	2m42.41s	50.43 MB	0.10, 13^{th} Epoch

TABLE 1. FCN 100K Configurations and Test Results. We implemented two layers of FCN with 100 and 180 neurons in each layer respectively amounting to total of 98,490 trainable parameters (including both weights and bias), and used Adam optimizer with learning rate of 0.001 and batch normalization in both cases.

4.2. **FCN Variants with 50K and 200K Weight Constraints.** We furthered our experiments with 50K and 200K weight constraints for FCN, results are shown in Table 2. The results match with our intuition that size of Multiply and Add operations increases linearly with the number of contraints; and higher budgets will yield higher accuracy. In our cases, 200K model doubles the operations of 100K which in turn doubles 50K; 200K model receives highest accuracy of 88.7988% among the three, and 50K has the lowest accuracy (87.4219%). Similarly, GPU benefits only renders in cases with larger number of operations, e.g. 200K GPU runtime slightly faster than 50K. Underfitting can be observed in all three cases with 50K being the most undertrained as the loss could potentially go closer to 0.

FCN	Weights	Test Acc.	CPU-only	with GPU	Mult.Adds	Low.Loss
50K	46,580	87.4219%	2m 25.98s	2m 43.99s	23.85 MB	0.16, 19^{th} Epoch
100K	98,490	88.5156%	2m 32.93s	2m 42.66s	50.43 MB	0.11, 17^{th} Epoch
200K	199,210	88.7988%	2m 36.82s	2m 42.94s	102.00 MB	0.08, 19^{th} Epoch

TABLE 2. FCN 50K, 100K, and 200K Configurations and Test Results. Run Times for CPU-only and CUDA-enabled computations differ slightly, the Single Instruction Multiple Data (SIMD) benefits minimally potentially due to latencies generated in read and write-backs from host to device.

4.3. **CNN 100K Configurations.** After numerous experiments with number of layers and numbers of channels in each layer, we found 2 layers of 2-dimensional convolutional network with channels of 1 (greyscale input), 40 (first convolutional layer), and 60 (second convolutional layer, output to FC layers) added before a 3-FC layers of 25 (first FC layer, input from 2nd convolutional layer), 50 (second FC layer), 10 (third FC layer, output to classified prediction) dimensions meet our expectation of desired accuracy (90.7031%) and weights constraint (98,395 total trainable weights). With our implementation of CNN, we experienced huge benefits in using high performance parallel computing as great improvements of runtime in GPU were observed when operation numbers reached 2GBs, see the following Table 3 for configurations. CNN 100K layers can easily

[1]Due to stochastic nature of batch split and training algorithm, the specified test accuracy and train loss values in Section 4 and figures in Appendices were all generated in the CPU-only experiments. GPU experiments results were used mainly for runtime comparisons.

achieve test accuracy above 90% which in our last report experienced hardship in training FCN to reach the same accuracy.

Current FCN 100K has a more straight forward comparison with CNN 100K as both train the same amount of parameters, same amount of runtime if CUDA-enabled, but the former only has accuracy of 88.52% compared with 90.70% for the later. But the size of operations increases 50x (50MB v. 2.4GB) for CNN to manifest that 2.5% accuracy boost.

CNN	**Weights**	**Test Acc.**	**Conv.Channels**	**Padding**	**Conv.Ker.**	**Pool.Ker.**
100K	97,395	90.7031%	1, 40, 60	1	3	2
FC Dim.	**Opt.**	**L.R.**	**CPU-only**	**with GPU**	**Mult.Adds**	**Low.Loss**
20, 50, 10	Adam	0.001	23m 51.10s	2m 58.46s	2.37 GB	0.03, 20^{th} Epoch

TABLE 3. CNN 100K Base Configurations to reach accuracy of 90.7031% with batch normalization for both convolutional and fully-connected layers, without Initialization or Dropout for the FC layers.

4.4. **CNN Variants with 50K, 20K, and 10K Weight Constraints.** We continued with our experiments on different weight budgets for CNN models. For the effectiveness of training outcome of the 2-D Convolutional layers, we keep 2 convolutional layers and adjust the number of neurons in 3 FC layers accordingly to our constraints. Specific configurations for each model are listed in Table 4. The most weights happen when we apply the first FC layer, therefore, we mainly adjust the number of channel of the last convolutional layer and the number of neurons in the first FC layer and induced our configurations.

CNN	**Weights**	**Conv.Channels**	**FC Dimensions**
10K	9,192	1, 10, 18	8, 20, 10
20K	19,279	1, 20, 30	9, 20, 10
50K	49,450	1, 25, 40	20, 30, 10
100K	97,395	1, 40, 60	25, 50, 10

TABLE 4. CNN model Configurations with 10K, 20K, 50K, and 100K Weights.

The outcome surprises us in many aspect as shown in Table 5 below. We observed an easy above 90% accuracy even for 20K weights in CNN, whereas a 200K FCN will plateau at accuracy merely of 88.7988%. Moreover, the increasing size of operations increases CPU run-time, whereas CUDA-enabled computations benefits tremendously on SIMD with run-time almost stays the same (less than 3 mins). In addition, we observed a slight improvement in accuracy (from 90.7031% to 91.2207%) even when we halved our weights from 100K to 50K; also, if we only used 20K weights, our accuracy will still robustly stay at 90.7031%. Apart from the desired test accuracy, we still detected under-trained signals from the values of loss function. It fits our expectation that higher weights correspond to lower train loss value showing better converge trend, which also in concert with our observations on train loss values in FCN variants.

4.5. **CNN Feature Map Visualization.** We chose CNN 20K variant for two reasons: (1) comparably favorable accuracy above 90%, no inferior than CNN 100K; and (2) runtime efficiency especially for CPU-only circumstance when limited computational resources are available. We then applied a hook function to "hook" certain convolutional layer and stored the gradient values for visualization. We chose sample number 1 (boots), number 4(pants), and 7 (t-shirt) for visualization from different classes. For each sample, we derived two feature maps for each convolutional

CNN	Test Acc.	CPU-only	with GPU	Mult.Adds	Low.Loss
10K	89.2871%	6m 15.81s	2m 51.49s	208.33 MB	0.21, 19^{th} Epoch
20K	90.7031%	10m 34.19s	2m 52.56s	632.18 MB	0.13, 18^{th} Epoch
50K	91.2207%	13m 44.14s	2m 52.62s	1.03 GB	0.06, 20^{th} Epoch
100K	90.7031%	23m 51.10s	2m 58.46s	2.37 GB	0.03, 20^{th} Epoch

TABLE 5. CNN 10K, 20K, 50K, 100K Test Results and Runtime Comparison Between CPU-only and GPU. Favorable test accuracies ($> 90\%$) are obtained for all variants except 10K. GPU runtime differs slightly amount the variants and benefits greatly from parallel computing compared with CPU-only models. Number of operations have linear increase with the size of CNN weights.

layer respectively. We observed that for all three samples, the first convolutional layer generates clearer images (or, higher resolutions) (Figure 2) compared with the second layer, from which the generated features are more blurred (Figure 3). We speculate that this may due to the fact that the deeper convolutional layer it is, the less detailed feature it will remain. The deeper layer emphasize more on edges or large changes in the neighborhood. See Figure 2 and 3 in Appendix A.

5. Summary and Conclusions

We designed a FCN model with adjustable number of hidden layers, neurons in each layer, and other hyperparameters, a CNN model with adjustable convolutional layers with adjustable channels, kernel number, pooling function, with additional adjustable FC layers. We used Cross Entropy loss function for both models to perform classification on FashionMNIST dataset. We tried out different weight constraints (50K, 100K, 200K for FCNs, and 10K, 20K, 50K, 100K for CNNs), compared test accuracy and runtime for all variants. We concluded that CNN models in general performed better than same weight FCN models for the fact of number of operations involved in the training process. We observed tremendous efficiency boost in using CUDA-enabled GPU in computing CNN variants especially for operations in GB scale.

Acknowledgements

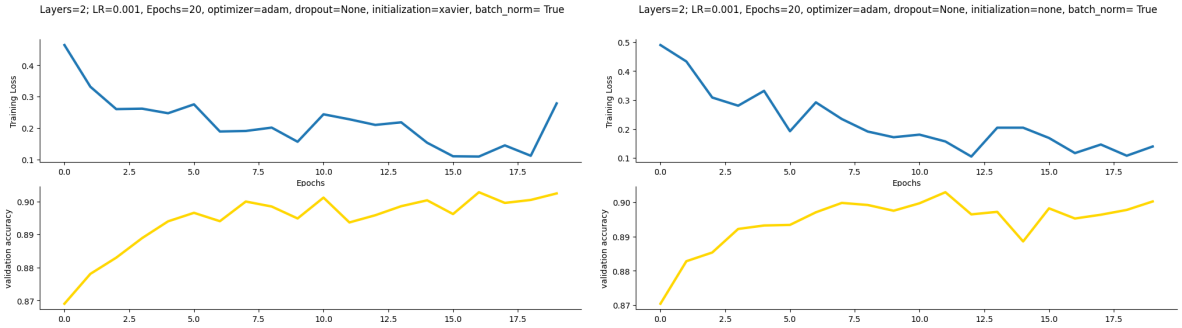
The author is thankful to Prof. Eli for useful discussions and code samples about Fully Connected Deep Neural Networks and Convolutional Neural Networks; and also thankful to the usage of computational resources on CUDA-enabled GPU model training provided by AWS cloud computing.

References

[1] I. J. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. MIT Press, Cambridge, MA, USA, 2016. http://www.deeplearningbook.org.

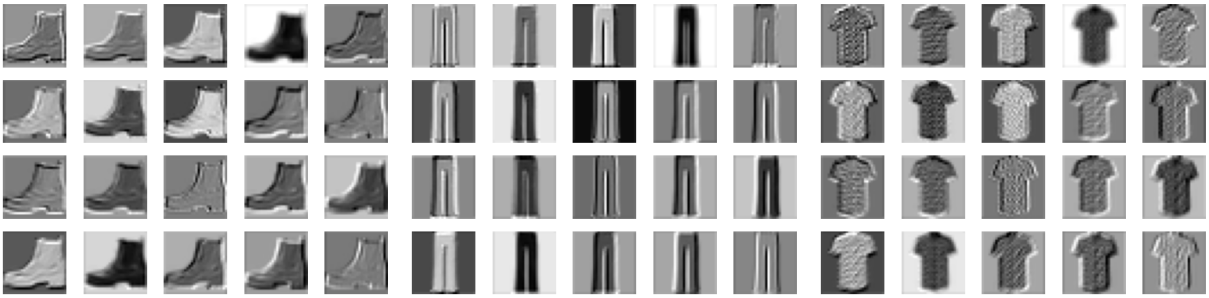
[2] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Köpf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala. Pytorch: An imperative style, high-performance deep learning library, 2019.

Appendix A. Figures



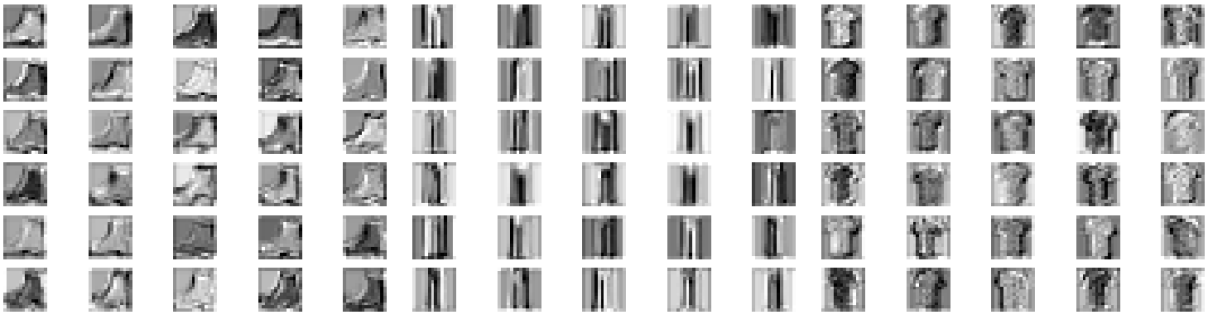
(A) FCN 100K Initialized with Xavier Normal (B) FCN 100K without Initialization

FIGURE 1. 1a Validation accuracy and Training loss curve initialized with Xavier Normal with learning rate of .001 using Adam Optimizer. 1b No initialization.



(A) Feature Map for Boots. (B) Feature Map for Pants. (C) Feature Map for T-Shirts.

FIGURE 2. Feature Map Generated from the First Convolutional Layer with Output Channel 20. 2a (left) Feature Map for Sample 1 (boots). 2b (middle) for Sample 4 (pants). 2c (right) for Sample 7 (T-Shirts).



(A) Feature Map for Boots. (B) Feature Map for Pants. (C) Feature Map for T-Shirts.

FIGURE 3. Feature Map Generated from the Second Convolutional Layer with Output Channel 30. 3a (left) Feature Map for Sample 1 (boots). 3b (middle) for Sample 4 (pants). 3c (right) for Sample 7 (T-Shirts).