Project I
Latte

Shao Yi

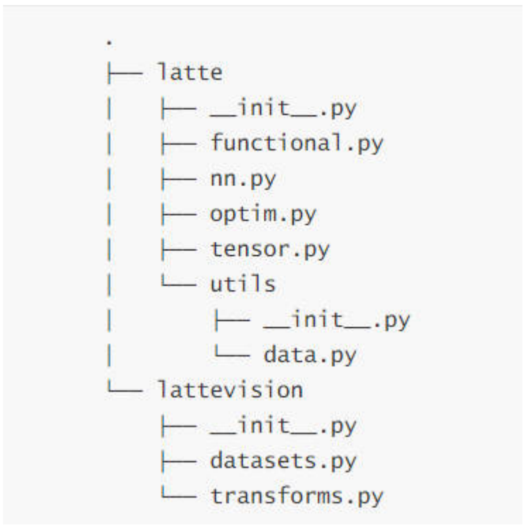
2022-04-09

Abstract

Latte (**L**et's **A**bsorb **T**orch **T**echnology **E**legantly) is a self-designed deep learning framework working on CPU, the package name shows tribute to Caffe while the inner structure is inspired by the PyTorch framework. This project focuses on the manual implementation of the most common deep learning package modules and solves the **MNIST** dataset classification problem.

1 Introduction

The package structure in the tree view is as follows:



```
.
├── latte
|   ├── __init__.py
|   ├── functional.py
|   ├── nn.py
|   ├── optim.py
|   ├── tensor.py
|   └── utils
|       ├── __init__.py
|       └── data.py
└── lattevision
    ├── __init__.py
    ├── datasets.py
    └── transforms.py
```

Figure 1: Package Structure

Latte packages some basic modules, like **tensor** defines the basic data structure, **nn** together with **functional** provides necessary classes and functions for neural network, **optim** includes some classical optimization methods, and **utils.data** is for dataset and data loader. All the interfaces are referred to the official documentation of PyTorch.

Furthermore, what torchvision is to torch, lattevision is to latte. It includes some requisite modules for computer vision tasks. To be more specific, **transforms** contains some transformation functions for image processing, **datasets** implements the downloading and loading of MNIST dataset for now.

All the features above are implemented using **numpy** package.

1.1 Dataset

Imitating the PyTorch **torchvision.datasets** module, we implement **MNIST**, the core code is shown below, for more details please refer to source code.

```python
class MNIST(VisionDataset):
    def __init__(
        self,
        root: str,
        train: bool = True,
        transform: Optional[Callable] = None,
        target_transform: Optional[Callable] = None,
    ) -> None:
        super().__init__(root, train, transform, target_transform)

    def _load_dataset(self) -> None:
        url = 'http://yann.lecun.com/exdb/mnist/'
        if self.train:
            data_filename = 'train-images-idx3-ubyte.gz'
            labels_filename = 'train-labels-idx1-ubyte.gz'
        else:
            data_filename = 't10k-images-idx3-ubyte.gz'
            labels_filename = 't10k-labels-idx1-ubyte.gz'

        data_filepath = get_data(url + data_filename, self.root)
        labels_filepath = get_data(url + labels_filename, self.root)

        self.data = self._load_data(data_filepath)
        self.labels = self._load_labels(labels_filepath)

    def _load_data(self, filepath: str) -> np.ndarray:
        with gzip.open(filepath, 'rb') as f:
            data = np.frombuffer(f.read(), np.uint8, offset=16)
        data = data.reshape(-1, 1, 28, 28)
        return data

    def _load_labels(self, filepath: str) -> np.ndarray:
        with gzip.open(filepath, 'rb') as f:
            labels = np.frombuffer(f.read(), np.uint8, offset=8)
        return labels
```

Therefore, all the network training is based on the official MNIST dataset instead of our 16*16 version.

1.2 Computational Graph

In the **tensor** module, building computational graph is wrapped in the backward function of **Tensor** class in order to support the backpropagation, the corresponding code is as follows:

```python
class Tensor:
def __init__(
        self,
        data: np.ndarray,
        grad_fn: 'Function' = None,
        requires_grad: bool = False,
        is_bias: bool = False,  # Notation for bias, not official
    ) -> None:
        pass

    # More specifications can be found in the source code

    def backward(self) -> None:
        # Build computational graph
        graph = []
        visited = set()

        def build_graph(node: 'Tensor'):
            if node.requires_grad is True and node not in visited:
                visited.add(node)

                # Post-order traversal
                if node.grad_fn is not None:
                    for prev_node in node.grad_fn.prev:
                        build_graph(prev_node)

                graph.append(node)

        build_graph(self)

        # Backpropagate gradients
        self.grad = np.array([1.0]).reshape(1, 1)  # Create implicit gradient
        for node in reversed(graph):
            if node.grad_fn is not None:
                node.grad_fn.backward(node.grad)
```

Besides, every single **basic operation** in our arithmetic library is implemented with the **backward function**. In this way, we can basically reproduce PyTorch's propagation mechanism.

1.3 Quick Start

This part is a quick start guide, we run a toy example to see how the framework works. By the way, all the code implementations are included in jupyter notebooks.

The nerwork architecture is shown in the following figure:

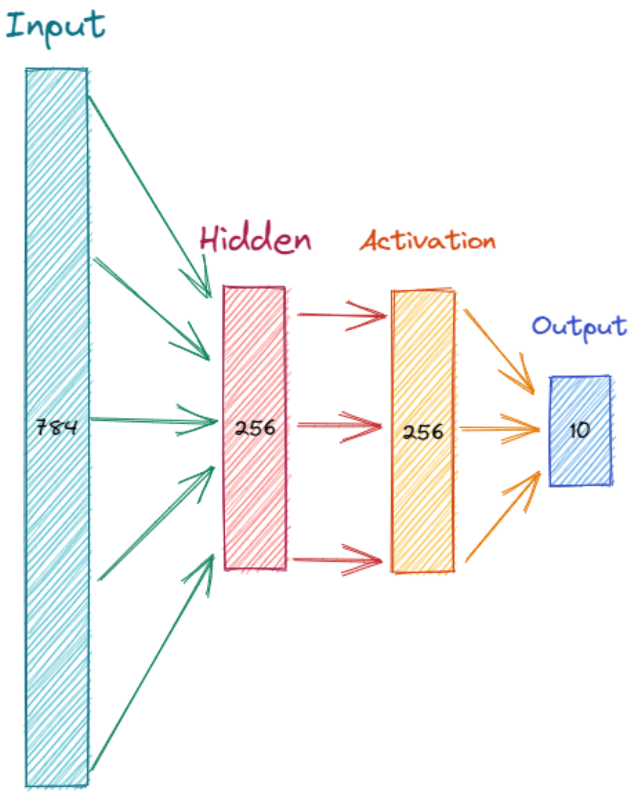


Figure 2: Simple Network

As an example, we use only **fully connected** layers to build a simple network, the activation function is **ReLU**. For criterion and optimizer, we use **CrossEntropyLoss** and **Adam** respectively.

The training result should be good because we already apply some advanced techniques in this example, so we just set training epochs to 5 and learning rate so as to 0.001 to see if the framework works. And here is the visualization of training process:

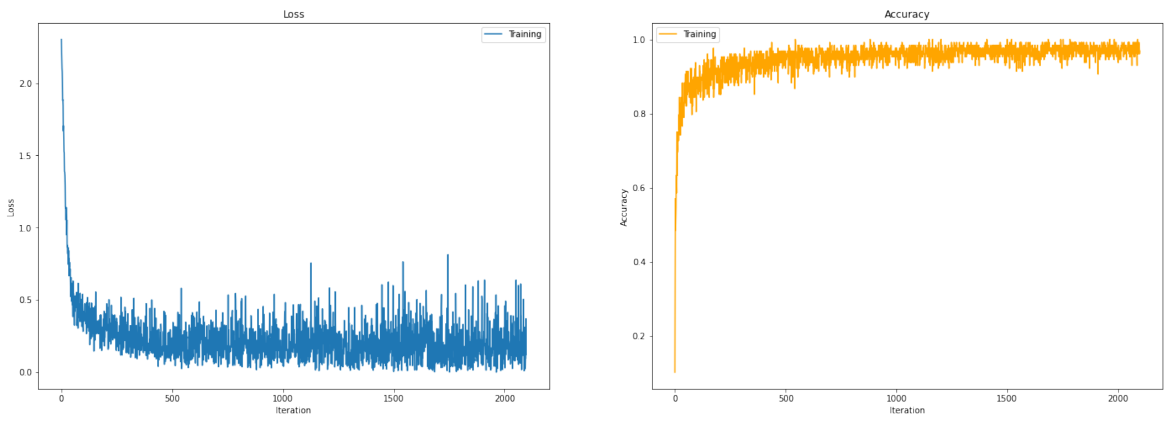


Figure 3: Loss and Accuracy

Finally we test the network on the test dataset and get the accuracy **97.01%**.

2 Modification

This part is some attempts to modify the original network architecture with some insights from the assignment document, such as momentum, dropout, data augmentation, convolution layers, etc. The code implementations are also in the jupyter notebook.

2.1 Change the Network Structure

In this section, we focus on both number of hidden units and number of hidden layers. We first fix the number of hidden layers to 1, and then change the number of hidden units accordingly. To avoid the overfitting, we test the network **every epoch**. Here is the result:

Units	7	8	9	10	11	12	13	14	15
64	0.9703	0.9718	0.973	0.9744	0.974	0.9753	0.9762	0.9767	**0.9767**
128	0.9695	0.972	0.9738	0.9741	0.9738	0.975	**0.9756**	0.9738	0.9754
256	0.9696	0.9706	0.9736	0.9743	0.9734	0.9736	0.9734	**0.9758**	0.9744
512	0.9686	0.9707	0.9708	0.9718	0.9725	**0.9738**	0.9735	0.9737	0.9731

Table 1: Number of Hidden Units

We can see that there is no significant difference between the accuracy of the network with 64, 128, 256 and 512 hidden units. One thing need to be noted is that network with 64 hidden units is not able to converge to the best accuracy due to the limited max epochs. Consequently, maybe there's **no monotonous relationship** between the number of hidden units and final accuracy.

Secondly, we try the network with 2 hidden layers, and units for each layer is 256, 64 in order.

6	7	8	9	10	11	12	13	14	15
0.9527	0.9586	0.9661	0.965	0.967	0.9698	0.9719	0.9733	0.9734	**0.9741**

Table 2: Deeper Network

Same situation as the 64 hidden units network, the max epochs for this network is not enough. However, as trying more training epochs, we encountered the problem of **dividing by zero** in log softmax, this is weird because it never happens before.

To put it into a nutshell, the network structure has **no significant impact** on the accuracy in MNIST classification task, at least the relationship between them is not simply ”the more complex the better”. And when it comes to more complicated network, you may meet other problems such as not enough training epochs.

2.2 Momentum in SGD

Momentum is a technique that can help the network to converge faster. In this section, we try momentum to train the network. The core code is in the following code snippet:

```python
class SGD(Optimizer):
    """SGD with momentum."""

    def __init__(
        self, params: List['Tensor'], lr: float = 1e-3, momentum: float = 0.9
    ) -> None:
        super().__init__(params, lr)
        self.momentum = momentum
        self.v = [np.zeros(param.shape) for param in self.params]

    def step(self) -> None:
        for param, v in zip(self.params, self.v):
            v = self.momentum * v + self.lr * param.grad
            param.data = param.data - v
```

Thus by setting the parameter momentum to 0 or 0.9, we experiment the effect of momentum in SGD optimizer. Here is the result:

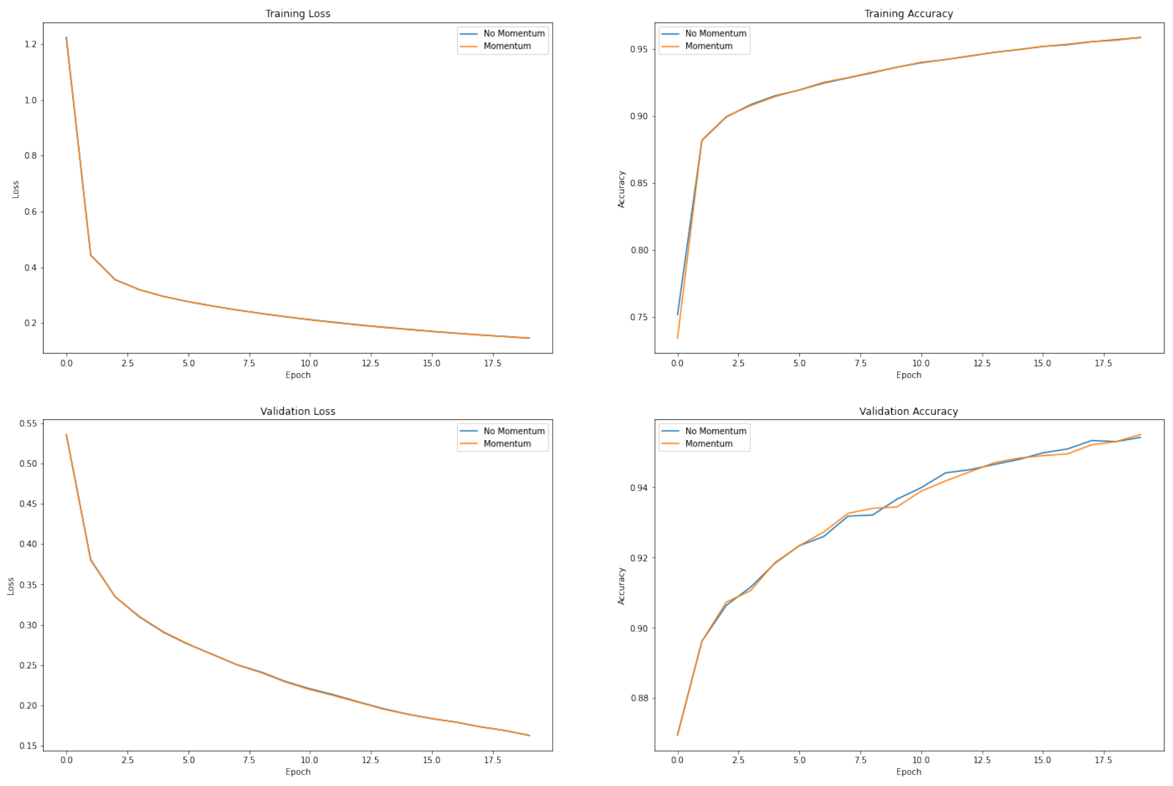


Figure 4: Momentum Effect

The test accuracy is almost equal (**95.72%** with momentum 0.9 and **95.77%** without momentum). We know momentum **absorbs the gradient** of the previous step, which means the network will have **inertia** in converging. Therefore, this inertia may not be a good thing in preliminary stage but when the network is in the final stage, this inertia definitely **accelerate the convergence**.

2.3 Vectorization

Vectorization is a classical technique to accelerate the matrix computation. As our whole framework is built on numpy, we already implement vectorization implicitly. So we can skip this section, just keep in mind that vectorization is faster than for loops, splendidly!

2.4 Regularization & Early Stopping

Regularization is easy to implement, and it is a good idea to avoid overfitting. However, before implementing regularization, we decide to plot the weights of toy model in heatmap first to see whether there are outliers.

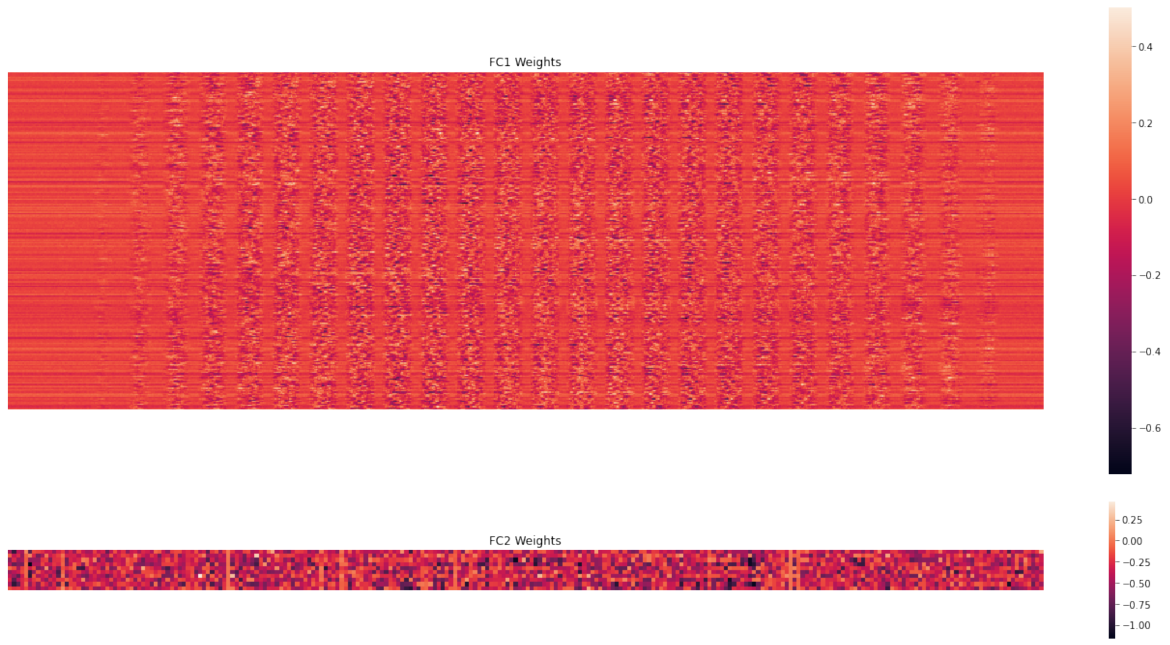


Figure 5: Heatmap of Weights

As we can observe, the **range of weights** meets the expectation, and there is no outlier.

The code snippet for regularization is as follows:

```python
class SGD(Optimizer):
    """SGD with momentum."""

    def __init__(
        self,
        params: List['Tensor'],
        lr: float = 1e-3,
        momentum: float = 0.9,
        weight_decay: float = 0.0,
    ) -> None:
        super().__init__(params, lr, weight_decay)
        self.momentum = momentum
        self.v = [np.zeros(param.shape) for param in self.params]

    def step(self) -> None:
        for param, v in zip(self.params, self.v):
            v = self.momentum * v + self.lr * param.grad

            if not param.is_bias:
                param.data = (
                    param.data
                    - v
                    - self.weight_decay
                    * param.data
                    / np.linalg.norm(param.data, ord='fro')
                )
            # Bias is not regularized
            else:
                param.data = param.data - v


class Adam(Optimizer):
    def __init__(
        self,
        params: List['Tensor'],
        lr: float = 1e-3,
        weight_decay: float = 0.0,
        beta1: float = 0.9,
        beta2: float = 0.999,
        eps: float = 1e-8,
    ) -> None:
        super().__init__(params, lr, weight_decay)
        self.beta1 = beta1
        self.beta2 = beta2
        self.m = [np.zeros(param.shape) for param in self.params]
        self.v = [np.zeros(param.shape) for param in self.params]
        self.t = 0  # number of steps
        self.eps = eps

    def step(self) -> None:
        self.t += 1
        lr_t = self.lr * np.sqrt(1 - self.beta2 ** self.t) / (1 - self.beta1 ** self.t
    )
        eps = self.eps * self.t ** 0.5
        for param, m, v in zip(self.params, self.m, self.v):
            m = self.beta1 * m + (1 - self.beta1) * param.grad
            v = self.beta2 * v + (1 - self.beta2) * param.grad ** 2

            if not param.is_bias:
                param.data = (
                    param.data
                    - lr_t * m / (np.sqrt(v) + eps)
                    - self.weight_decay
                    * param.data
                    / np.linalg.norm(param.data, ord='fro')
                )
            # Bias is not regularized
            else:
                param.data = param.data - lr_t * m / (np.sqrt(v) + eps)
```

After attempting the regularization with setting the weight decay to 0.01, the test accuracy **doesn't improve much**, this result is not surprising since the heatmap shows that the weights are not too far from the mean.

Next is early stopping, the code snippet is simple as follows:

```
# Early stopping
trigger_times = 0
patience = 2

```

```python
max_epochs = 10
train_losses = []
val_losses = [100]  # Initialize with a huge loss

for epoch in range(max_epochs):
    # Training stage (skip)

    # Validation stage(skip)

    # Early stopping
    if val_losses[-1] > val_losses[-2]:
        trigger_times += 1
        if trigger_times == patience:
            print("Early Stopping!")
            print(f'\tTraining Losses: {train_losses}')
            print(f'\tValidation Losses: {val_losses[1:]}')
            break
    else:
        trigger_times = 0
```

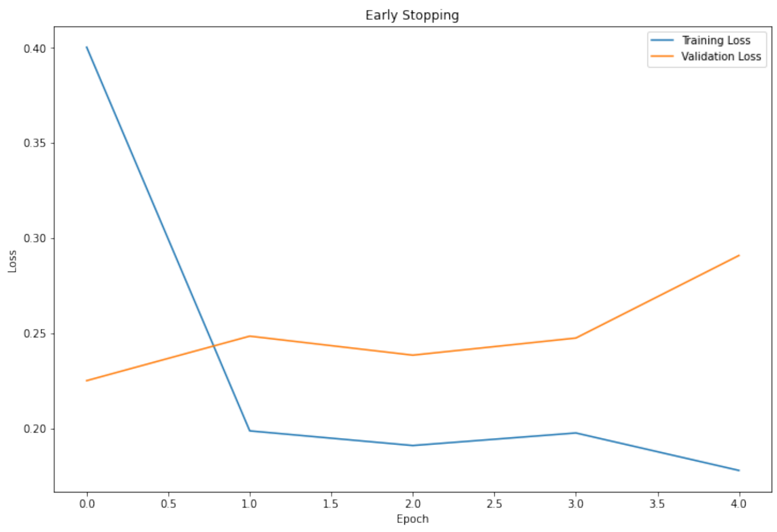


Figure 6: Early Stopping

The training process stops when the validation loss is increasing for a certain number of epochs, here we set default to 2. And as we can see from the figure, early stopping appears at 5th epoch out of 10 epochs. The test result is **96.97%**, not bad!

2.5 Softmax

Referring to PyTorch's nn.CrossEntropyLoss, we combine **log softmax** and **negative log likelihood loss** in a single function as follows:

```python
class CrossEntropyLoss(Function):
    """Combines log_softmax and nll_loss in a single function."""

    def __repr__(self) -> str:
        return 'Function(CrossEntropyLoss)'

    def forward(self, input: 'Tensor', target: 'Tensor') -> 'Tensor':
        self.save_backward_node([input, target])
        m = input.shape[0]
        input_data = input.data
        target_data = target.data

        # softmax = exp(x[target]) / sum(exp(x[i]), axis=1)
        neg_log_softmax = -input_data[np.arange(m), target_data] + np.log(
            np.sum(np.exp(input_data), axis=1)
        )

        return Tensor(
            np.mean(neg_log_softmax), grad_fn=self, requires_grad=input.requires_grad
        )

    def backward(self, out: np.ndarray) -> None:
        a, b = self.prev
        m = a.shape[0]
        input_data = a.data
        target_data = b.data

        # neg_log_softmax' = softmax - 1 * (i == target)
        softmax = np.exp(input_data) / np.sum(np.exp(input_data), axis=1).reshape(-1, 1)
        softmax[np.arange(m), target_data] -= 1
        a.grad = softmax / m
```

By training models in identical structure with same data batchs, we can fairly compare the performance of different loss functions. By the way, in order to amplify the difference, we use SGD instead of Adam.

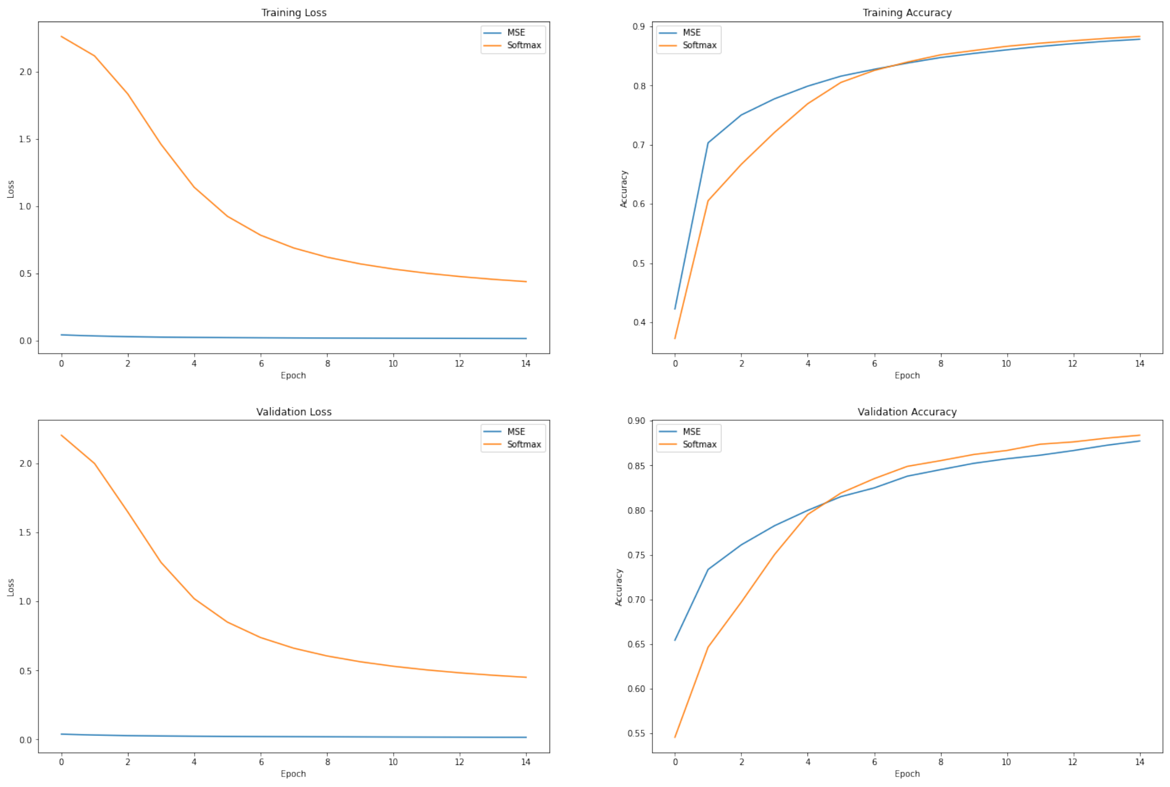


Figure 7: Loss Function Comparison

And the final results for MSELoss and CrossEntropyLoss are close too (**89.25%** and **89.23%** respectively). Although they are both in their way to converge, we can still find some macro laws, like the **training loss** of CrossEntropyLoss is much greater than that of MSELoss, which could be verified by simple examples. In this situation, CrossEntropyLoss would gain **greater gradient** than MSELoss, so as to make the training process more efficient.

So we just claim that **CrossEntropyLoss** is a better loss function for classification tasks.

2.6 Bias

The linear layer module is designed to have the interface of bias in advance, like:

```python
class Linear(Module):
    def __init__(self, in_features: int, out_features: int, bias: bool = True) -> None
    :
        super().__init__()
        self.weight = Tensor(
            np.random.randn(in_features, out_features) * 0.01, requires_grad=True
        )

        if bias:
            self.bias = Tensor(np.zeros((1, out_features)), requires_grad=True)
            self._params.extend([self.weight, self.bias])
        else:
            self.bias = None
            self._params.append(self.weight)

    def __repr__(self) -> str:
        return f'Linear({self.weight.shape[0]}, {self.weight.shape[1]}, \
            bias={self.bias is not None})'

    def forward(self, input: 'Tensor') -> 'Tensor':
        if self.bias is not None:
            return input.dot(self.weight) + self.bias  # x @ W + b
        else:
            return input.dot(self.weight)  # x @ W
```

Here follows the comparison of whether bias or not in the linear layer module. In addition, we choose SGD optimizer to slow down the convergence, intended to make the training process more detailed.

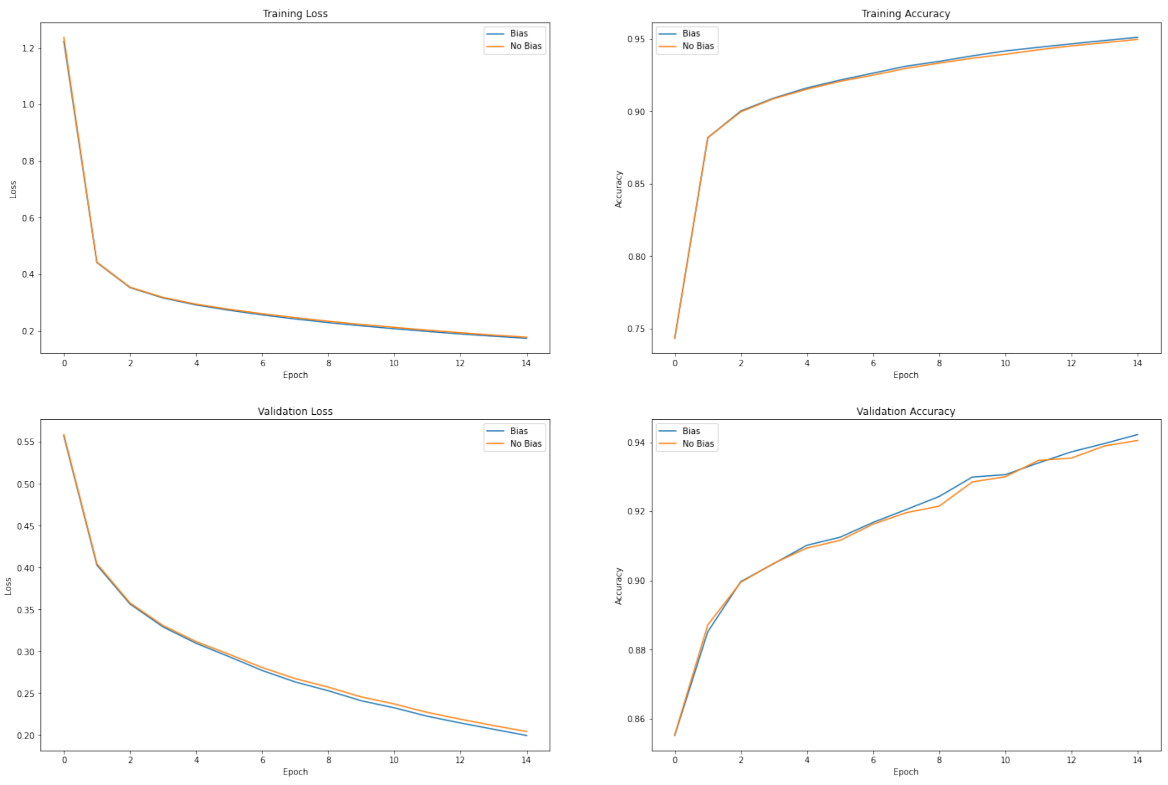


Figure 8: Bias Effect

As we can see, they behave similarly and ultimately come up with same result, in test dataset they perform accuracies **95.24%** (with bias) and **95.00%** (without bias), very closely.

Consequently we guess the **bias is not a big deal** in this situation.

2.7 Dropout

During training, the Dropout layer **randomly zeroes** some of the elements of the input tensor **with probability p** using samples from a Bernoulli distribution. Each channel will be zeroed out independently on every forward call. Furthermore, the outputs are **scaled** by a factor of $\frac{1}{1-p}$.

```python
class Dropout(Function):
    def __init__(self, p: float = 0.5, training: bool = True) -> None:
        super().__init__()
        self.p = p
        self.training = training
        self.mask = None

    def __repr__(self) -> str:
        return 'Function(Dropout)'

    def forward(self, a: 'Tensor') -> 'Tensor':
        if self.training:
            self.save_backward_node([a])
            out = a.data
            self.mask = np.random.binomial(1, self.p, size=a.shape) / (1 - self.p)
            out = out * self.mask
            return Tensor(out, grad_fn=self, requires_grad=a.requires_grad)

        else:
            return Tensor(a.data, grad_fn=self, requires_grad=a.requires_grad)

    def backward(self, out: np.ndarray) -> None:
        a = self.prev[0]
        a.grad = self.mask * out
```

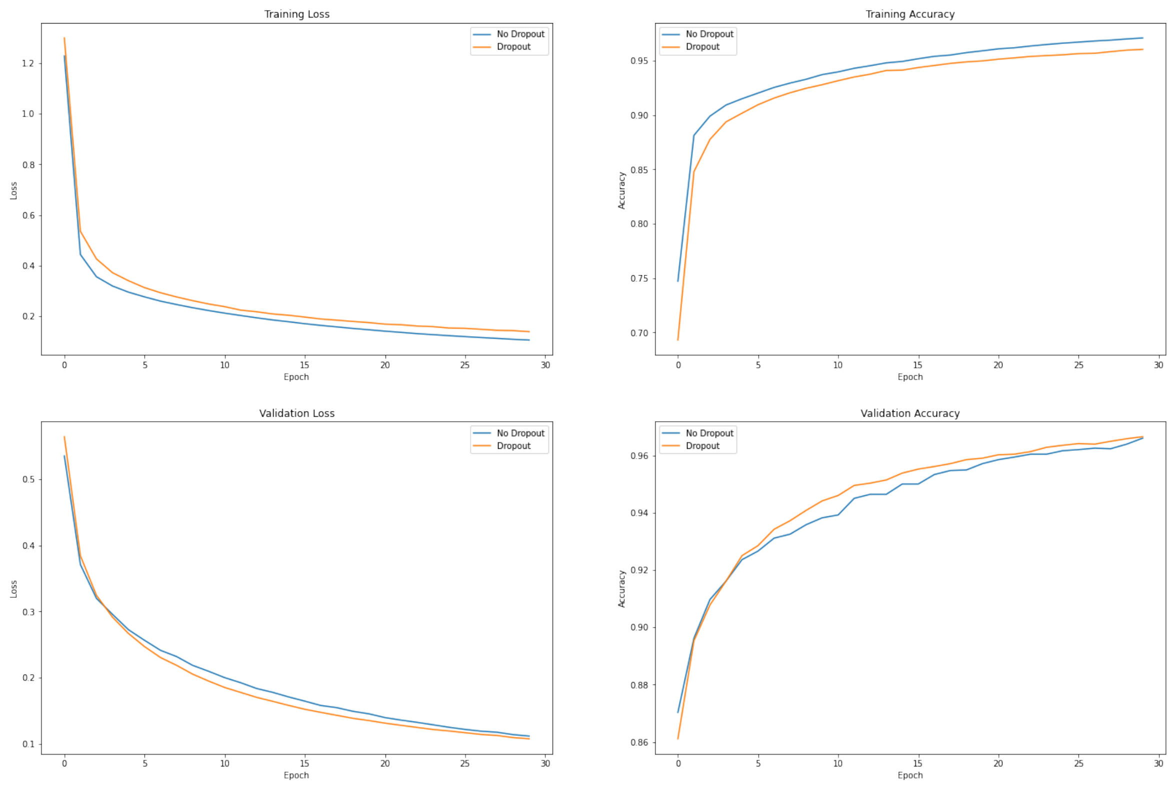


Figure 9: Dropout Effect

The effect of Dropout is that the model is less sensitive to the noise and the training process is more **stable**. However, experiments show that Dropout **helps little** in this task (We have tried $p = 0.2, 0.5, 0.7$, figure above is case of $p = 0.5$). The accuracies are **96.73%** (with Dropout) and **96.67%** (without Dropout).

2.8 Fine Tuning

Fine tuning is tailor-made for **MSELoss**, our code implements like this:

```
# * Fine tuning
fc2_bias_data = model_finetune.fc2.bias.data
x_ft = np.concatenate(x_ft)
y_ft = np.concatenate(y_ft)
x_ft_mpinv = np.matmul(
    np.linalg.pinv(np.matmul(x_ft.T, x_ft)), x_ft.T
)  # Moore-Penrose pseudoinverse
fc2_weight_data = np.matmul(
    x_ft_mpinv, (y_ft - fc2_bias_data)
)  # (X^T * X)^-1 * X^T * (Y - b)
model_finetune.fc2.weight.data = fc2_weight_data
```

However this time, we are pleasantly surprised by the result. The accuracy of the model with fine tuning is **97.11%** while the other without fine tuning is only **95.83%**, seems fine tuning really helps **improve the converging speed**.

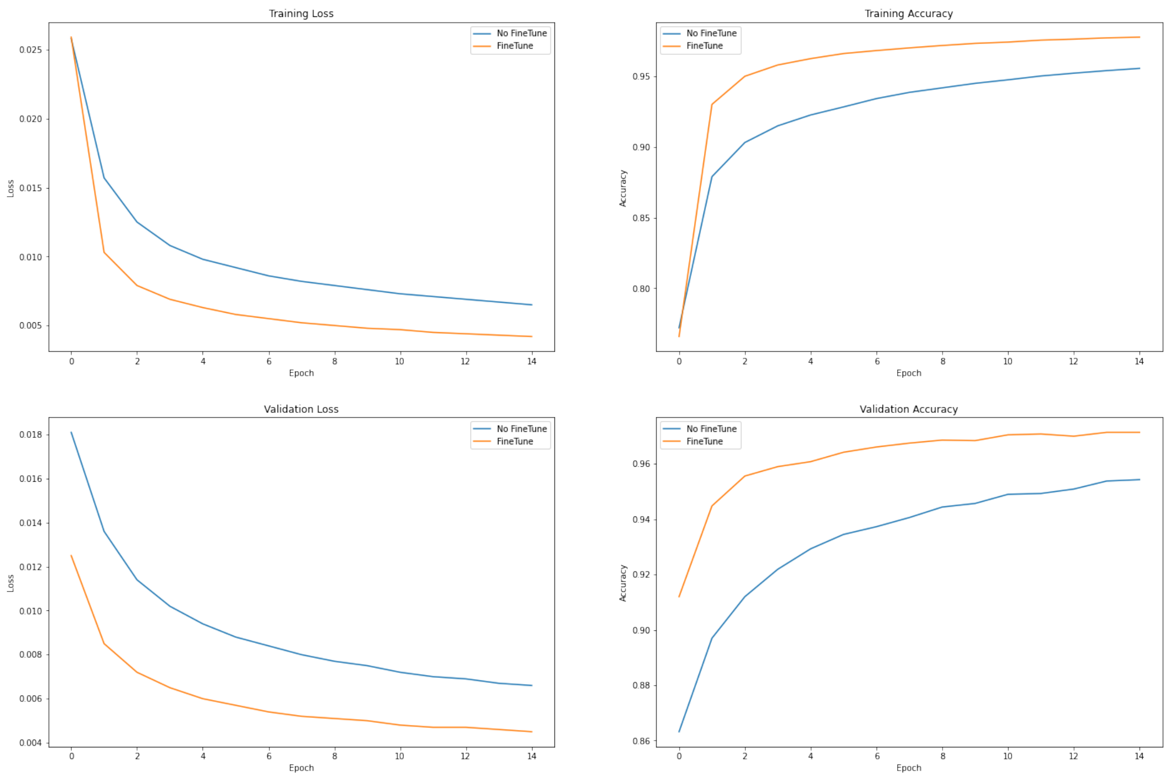


Figure 10: Fine Tuning

2.9 Data Augmentation

Referring to torchvision.transforms, we implement data augmentation like normalize, resize, crop, rotate, etc.

```
size = 32  # Format support: (h, w) or k for (k, k)
```

```
mnist_transform = T.Compose([T.Resize(size), T.ToTensor(), T.Normalize((0.1307,),
    (0.3081,))])
mnist_train = dsets.MNIST(mnist_root, train=True, transform=mnist_transform)
```

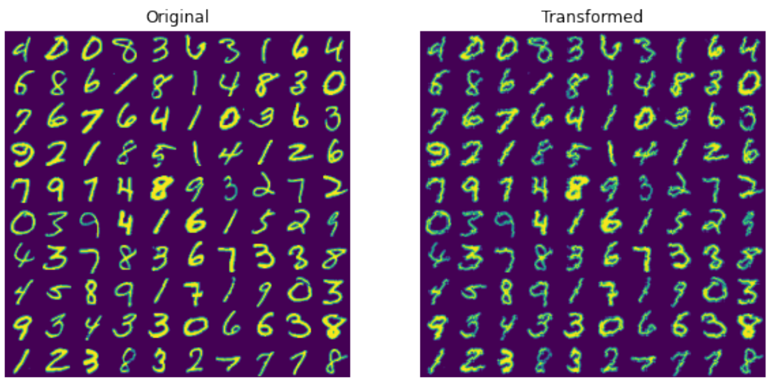


Figure 11: Resize Illustration

```
size = 16  # Format support: (h, w) or k for (k, k)
mnist_transform = T.Compose([T.CenterCrop(size), T.ToTensor(), T.Normalize((0.1307,),
    (0.3081,))])
mnist_train = dsets.MNIST(mnist_root, train=True, transform=mnist_transform)
```

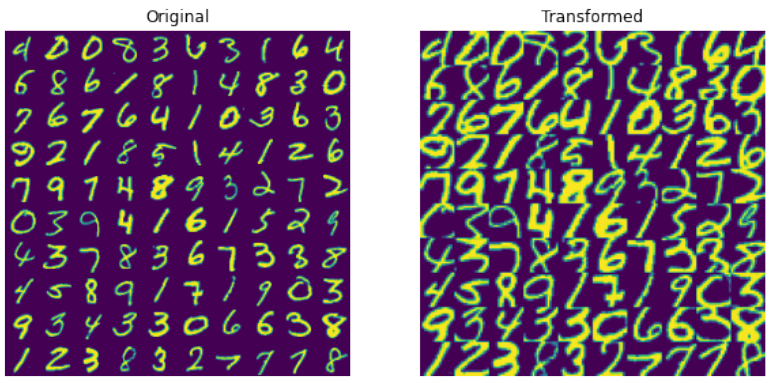


Figure 12: CenterCrop Illustration

```
size = 20  # Format support: (h, w) or k for (k, k)
mnist_transform = T.Compose([T.RandomResizedCrop(size), T.ToTensor(), T.Normalize
    ((0.1307,), (0.3081,))])
mnist_train = dsets.MNIST(mnist_root, train=True, transform=mnist_transform)
```

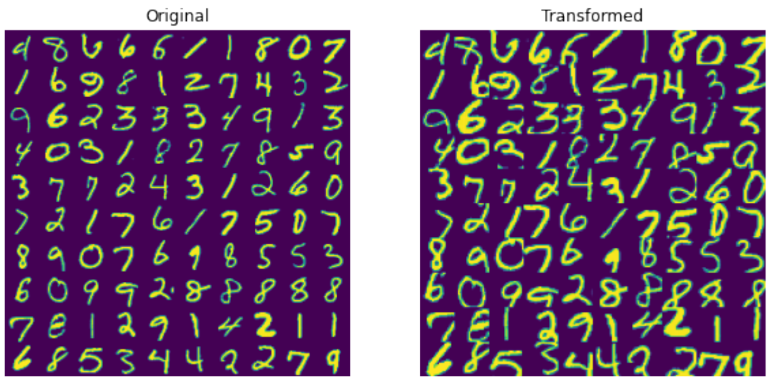


Figure 13: RandomResizedCrop Illustration

```
radians = (-np.pi / 4, np.pi / 4)  # Conter-clockwise is positive
mnist_transform = T.Compose([T.RandomRotation(radians), T.ToTensor(), T.Normalize
    ((0.1307,), (0.3081,))])
mnist_train = dsets.MNIST(mnist_root, train=True, transform=mnist_transform)
```

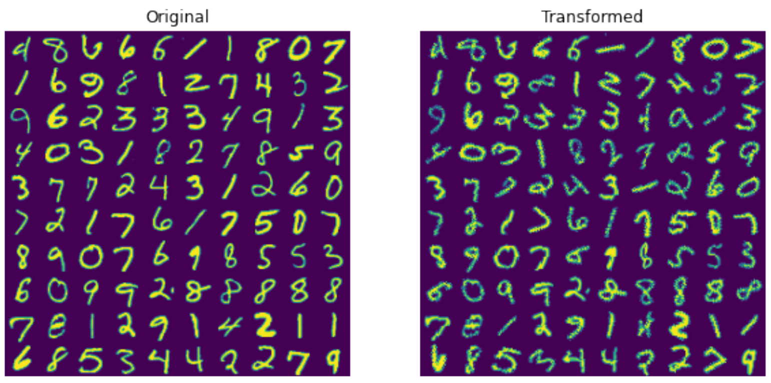
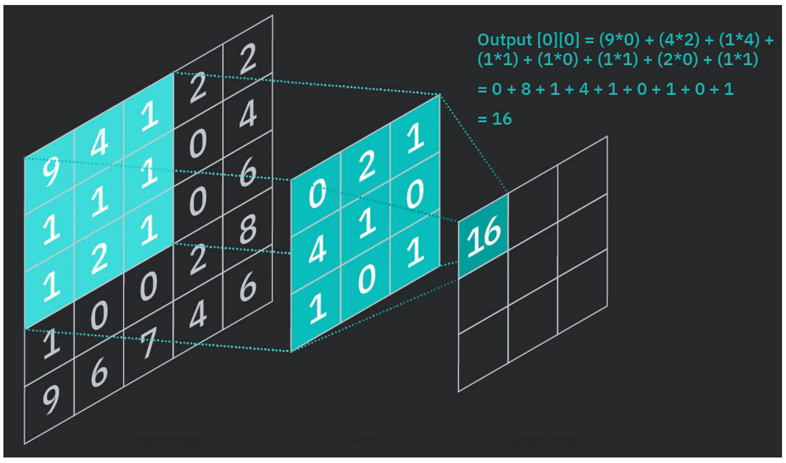


Figure 14: RandomRotation Illustration

After setting to different transforms, we find that the training result does not improve much. Maybe the original dataset is **already sufficiently complete**.

2.10 Convolutional Layer

Convolutional layer is one of the most powerful layers in deep learning, it can be used to **extract features** from the input data. An illustration of convolution operation is shown below:



Output [0][0] = (9*0) + (4*2) + (1*4) + (1*1) + (1*0) + (1*1) + (2*0) + (1*1)
= 0 + 8 + 1 + 4 + 1 + 0 + 1 + 0 + 1
= 16
16

Figure 15: Convolution Operation

Here is partial code of convolution operation:

```
class Conv2d(Module):
    def __init__(
        self,
        in_channels: int,
        out_channels: int,
        kernel_size: Tuple[int, ...],
        stride: Tuple[int, ...] = (1, 1),
        padding: Tuple[int, ...] = (0, 0),
        bias: bool = True,
    ) -> None:
        super().__init__()
        self.in_channels = in_channels
        self.out_channels = out_channels
        self.kernel_size = _pair(kernel_size)
        self.stride = _pair(stride)
        self.padding = _pair(padding)

        # Initialize weight and bias
        scale = np.sqrt(
            2.0 / (self.in_channels * self.kernel_size[0] * self.kernel_size[1])
        )
        self.weight = Tensor(
            np.random.randn(self.out_channels, self.in_channels, *self.kernel_size)
            * scale,
            requires_grad=True,
        )
        if bias:
            self.bias = Tensor(
                np.zeros((1, self.out_channels, 1, 1)),
                requires_grad=True,
                is_bias=True,
            )
            self._params.extend([self.weight, self.bias])
        else:
            self.bias = None
            self._params.append(self.weight)

    def forward(self, input: 'Tensor') -> 'Tensor':
        return F.conv2d(input, self.weight, self.bias, self.stride, self.padding)
```

And the result shows that network with convolutional layer can be trained much faster than network with only linear layer. The accuracies on test dataset are **97.21%** and **96.34%** respectively.

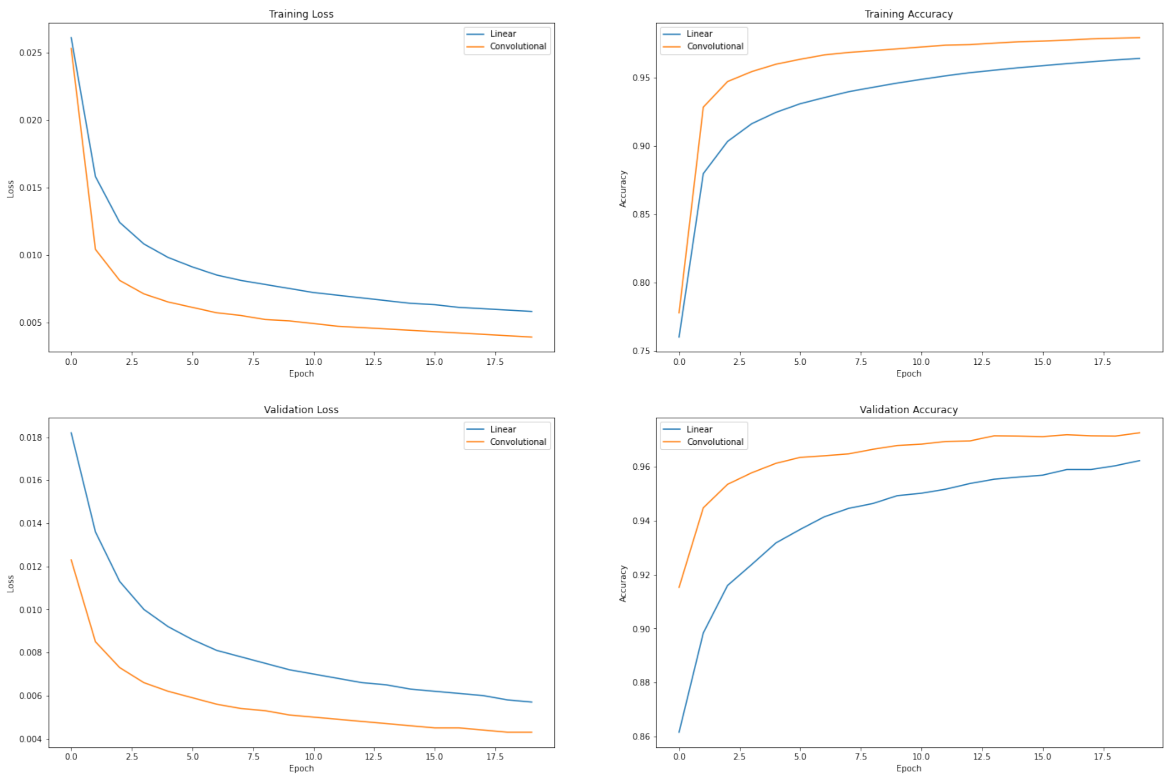


Figure 16: Convolutional Layer