2020 Pattern Recognition and Machine Learning Technical Report

Ruian He, 16307110216,

I. TASK DESCRIPTION

DESIGN a proper convolutional neural network for the classification task based on the MINIST dataset.[1]

II. DATA DESCRIPTION

The MNIST database of handwritten digits, available from this page, has a training set of 60,000 examples, and a test set of 10,000 examples. It is a subset of a larger set available from NIST. The digits have been size-normalized and centered in a fixed-size image. [3]

The data is stored in a very simple file format designed for storing vectors and multidimensional matrices.

A. Label File

The label file is like the following.The labels values are 0 to 9.

[offset]	[type]	[value]	[description]
0000	32 bit integer	0x00000801(2049)	magic number
0004	32 bit integer	60000	number of items
0008	unsigned byte	??	label
0009	unsigned byte	??	label
...	...	...	...
xxxx	unsigned byte	??	label

B. Image File

And the image file is as followed.Pixels are organized row-wise. Pixel values are 0 to 255. 0 means background (white), 255 means foreground (black).

[offset]	[type]	[value]	[description]
0000	32 bit integer	0x00000803(2051)	magic number
0004	32 bit integer	60000	number of images
0008	32 bit integer	28	number of rows
0012	32 bit integer	28	number of columns
0016	unsigned byte	??	pixel
0017	unsigned byte	??	pixel
...	...	...	...
xxxx	unsigned byte	??	pixel

III. DATA PREPROCESSING

First of all, we uncompress the `.gz` files to get ubyte files described as above. And we use struct module in python to extract the magic number, the image number, the row number and col number. Then we read $num*row*col$ numbers stored in unsigned bytes from the file which is the data. We can visualize using `matpoltlib` module.

We also can plot the average image for every digit, and get a general view over all training data.

[1]http://yann.lecun.com/exdb/mnist/.

Fig. 1. Sample Digit

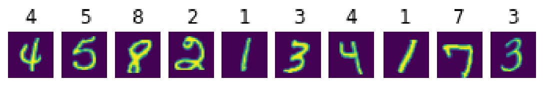
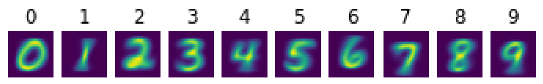


Fig. 2. Average Digit



IV. ALGORITHM INTRODUCTION

A. K-Nearest Neighbor

1) Algorithm principle: In pattern recognition, the k-nearest neighbors algorithm (k-NN) is a non-parametric method proposed by Thomas Cover used for classification and regression.In both cases, the input consists of the k closest training examples in the feature space. The output depends on whether k-NN is used for classification or regression.

In k-NN classification, the output is a class membership. An object is classified by a plurality vote of its neighbors, with the object being assigned to the class most common among its k nearest neighbors (k is a positive integer, typically small). If k = 1, then the object is simply assigned to the class of that single nearest neighbor. [4]

For specified train set and test set, we can easily calculate the distance between test point and every point in training set.We know in a large space the distance d_{ij} is as followed and k is the index of features.

$$\begin{aligned} d_{ij} &= \sqrt{\sum_{k=1}^{784}(x_{i,k} - x_{j,k})^2} \\ &= \sqrt{\sum_{k=1}^{784}(x_{i,k})^2 + \sum_{k=1}^{784}(x_{j,k})^2 - \sum_{k=1}^{784}2x_{i,k}x_{j,k}} \end{aligned}$$

2) Algorithm implementation: For training, we simply store all samples in the model. When predicting we calculate the distance between target point and all sample points.

We use `argsort` function in numpy module to find the points with the smallest k distance.The `argsort` function returns the indices that would sort an array. And use `argmax` and `bincount` function to find the label with the most votes from its neighbors. The `bincount` fuction count number of occurrences of each value in array of non-negative in.ts

Listing 1. K Nearest Neighbor

```
class KNearestNeighbor(object):
  def __init__(self):
    self.pre_X = None
    self.pre_dists = None

  def train(self, X, y):
    self.X_train = X
    self.y_train = y

  def predict(self, X, k=1):
    if (self.pre_X != X).any():
      self.pre_X = X
      self.pre_dists = self.
          compute_distances(X)
    return self.predict_labels(self.
        pre_dists, k=k)

  def compute_distances(self, X):
    num_test = X.shape[0]
    num_train = self.X_train.shape[0]
    dists = np.zeros((num_test, num_train
        ))
    dists = np.sqrt((self.X_train.dot(X.T
        )*(-2)+np.sum(np.square(X),axis=1)
        ).T+np.sum(np.square(self.X_train)
        ,axis=1))
    return dists

  def predict_labels(self, dists, k=1):
    num_test = dists.shape[0]
    y_pred = np.zeros(num_test)
    for i in range(num_test):
      closest_y = []
      closest_y=self.y_train[np.argsort(
          dists[i,:])][:k]
      y_pred[i]=np.argmax(np.bincount(
          closest_y))
    return y_pred
```

B. Bayesian Decision

1) Algorithm principle: In statistics, Naive Bayes classifiers are a family of simple "probabilistic classifiers" based on applying Bayes' theorem with strong (naïve) independence assumptions between the features.

When dealing with continuous data, a typical assumption is that the continuous values associated with each class are distributed according to a normal (or Gaussian) distribution.

Next, we start to build Bayesian Decision model from scratch. As the data follow Gaussion distribution, we can calculate the miu and sigma of the Gaussion distribution for every digit and we know the prior for each digit from the training set. [1]

Actually we can choose monovariable or multivariable gaussian distribution to describe. For the monovariable model, we assume that the features are irrelevant, and we can simply multiply the possibility $p_i(x_j)$ of every feature to get the likelihood L_i of every digit.

$$L_i = \prod_{i=1}^{784} p_i(x_j) = \prod_{i=1}^{784} \frac{1}{\sqrt{2\pi}\sigma_i} e^{-\frac{x_j-\mu_i}{2\sigma_i}}$$

As for multivariable model, we use covariance to replace variance in monovariable model to describible the connection between features. And we only need to calculate one possibility for one point. For the 2π term is same among all likelihood, we can ignore it when calculating the posterior.

$$L_i = p_i(x) = \frac{1}{(2\pi)^{\frac{n}{2}}|\Sigma|^2} e^{-\frac{1}{2}(x-\mu_i)^T\Sigma^{-1}(x-\mu_i)}$$

When we predict, according to the bayesian theorem, we can get $posterior = likelihood * prior/evidence$ and use log on both sides. As we only need to compare the relative size, we can ignore that evidence which is the same for all posteriors. Then the final one to compare is $\log(prior) + \log(likelihood)$.The digit class which get the largest posterior will be the choice.

2) Algorithm implementation: When implementing the algorithm, we first extract the prior, mean, variance and covariance for each digit, and predict the label by calculating the likelihood of each digit using the above formulas.

For some reason, the model is overfitting on test set, so we must add some hyperparameter to the variance in order to smooth the gaussian distribution we predicted. Just like $\sigma' = \sigma + smooth$.

Listing 2. Bayesian Decision
```
class BayesianDecision(object):
  def __init__(self):
    self.eps = 1e-5
    self.smooth = 1000

  def train(self, X, y):
    n_features = X.shape[1]
    self.prior = np.bincount(y)/y.shape
        [0]
    self.miu = np.zeros((10,n_features))
    self.var = np.zeros((10,n_features))
    self.cov = np.zeros((10,n_features,
        n_features))
    for i in range(10):
      select = X[np.where(y == i)]
      self.miu[i] = np.mean(select,axis
          =0)
      self.var[i] = np.var(select,axis=0)
           + self.smooth
      self.cov[i] = np.cov(select.T) +
          self.smooth

  def predict(self, X, mode):
    likelihood = np.zeros((10,X.shape[0])
        )
    for i in range(10):
    diff = X - self.miu[i]
    if mode == 'multi':
        det_sqrt = np.sqrt(np.linalg.det(
            self.cov[i]))
        likelihood[i] = np.log(np.exp(np.
            diag(-1/2*diff.dot(np.linalg.
            pinv(self.cov[i])).dot(diff.T)))
            /(det_sqrt+1e-5))
    else:
        possibility = np.exp(-1/2*np.square
            (diff)/self.var[i])/np.sqrt(self
            .var[i])

    likelihood[i] = np.sum(np.log(
        possibility),axis=1)
    posterior = likelihood + np.log(self.
        prior+self.eps)[:,np.newaxis]
    return np.argmax(posterior,axis = 0)
```

C. Support Vector Machine(SVM)

1) Algorithm principle: The objective of the support vector machine algorithm is to find a hyperplane in an N-dimensional space(N — the number of features) that distinctly classifies the data points. [4] We can define the hyperplane as follows:

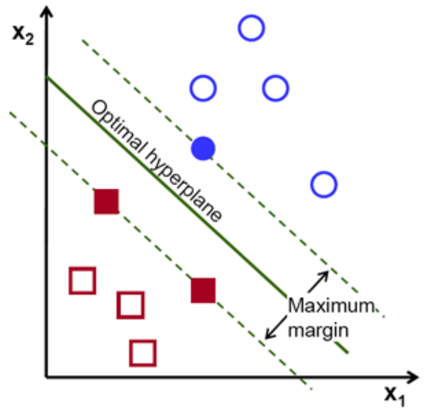
$$y = w^T x + b$$

To separate the two classes of data points, there are many possible hyperplanes that could be chosen. Our objective is to find a plane that has the maximum margin, i.e the maximum distance between data points of both classes. We define ρ_i as the distance of point i from the hyperplane.

$$\rho_i = \frac{1}{||w||} \cdot y_i(w^T x_i + b)$$

We introduce y_i in the distance because for positive examples $y_i = 1$ and we expect distance greater than a specific value $d_i = \frac{1}{||w||}(w^T x_i + b) \geq \delta$ and for negative examples $y_i = -1$ and and we expect distance $d_i = \leq -\delta$. We can unify this two presentations with ρ_i which y_i multiple $\frac{1}{||w||}(w^T x_i + b)$. And therefore $\rho_i \geq \delta$.

Fig. 3. Support Vector Machine



Our purpose is to maximize the minimum of ρ_i, i.e δ.

$$\max_{w,b} \delta$$
$$s.t. \frac{1}{||w||} \cdot y_i(w^T x_i + b) \geq \delta$$

Let $w' = \frac{w}{||w||\delta}$ and $b' = \frac{b}{||w||\delta}$, and our purpose becomes minimize $||w||$ so that $y_i(w'^T x + b') \geq 1$. And that is equivalent to minimize $\frac{1}{2}||w||^2$.

$$\min_{w,b} \frac{1}{2}||w||^2$$
$$s.t. y_i(w^T x_i + b) \geq 1$$

This is called the primal problem. It requires learning a large number of parameters in the w feature space.

The dual problem results in some beneficial properties that will aid in the computation, including the use of Kernel functions to solve non-linearly separable data.

The dual problem requires learning only the number of support vectors, which can be much fewer than the number of feature space dimensions. The dual problem is found by constructing the Lagrange, by combining both the objective function and the equality constraint function. The Lagrange formulation is

$$L(w, b, \alpha) = \frac{1}{2}||w||^2 - \sum_{i=1}^{n} \alpha_i (y_i(w^T x_i + b) - 1)$$

Here, we we define w and b as the primal variables and α_i as the dual variables. According to the Karush-Kuhn-Tucher(KKT) conditions, the formulation gets its optimal value when $\frac{\partial L(w,b,\alpha)}{\partial w} = \frac{\partial L(w,b,\alpha)}{\partial b} = 0$ and must fulfill the following conditions:

1) $\alpha_i \geq 0$
2) $y_i(w^T x_i + b) - 1) \geq 0$
3) $\alpha_i(y_i(w^T x_i + b) - 1) = 0$

Then we can get $w = \sum_{i=1}^{n} \alpha_i y_i x_i$ and $\sum_{i=1}^{n} \alpha_i y_i = 0$ from the above conditions.Finally, the formulation becomes like the following formula, and can be solved using Sequential Minimal Optimization(SMO) algorithm.

$$\min_{\alpha_i} L(\alpha) = \frac{1}{2} \sum_{i=1}^{n} \sum_{j=1}^{n} y_i y_j \alpha_i \alpha_j (x_i \cdot x_j) - \sum_{i=1}^{n} \alpha_i$$
$$s.t. \alpha_i \geq 0, \sum_{i=1}^{n} y_i \alpha_i = 0$$

In this formulation, $(x_i \cdot x_j)$ stands for the linear kernel function. We can replace this with other non-linear kernels like Radial Basis Function(RBF) to get better representation of features.

$$K(x_i, x_j) = \exp(-\frac{||x_i - x_j||^2}{2\sigma^2})$$

Algorithms such as the Perceptron, Logistic Regression, and Support Vector Machines were designed for binary classification and do not natively support classification tasks with more than two classes.

One approach for using binary classification algorithms for multi-classification problems is to split the multi-class classification dataset into multiple binary classification datasets and fit a binary classification model on each. Two different examples of this approach are the One-vs-Rest and One-vs-One strategies.

We choosed one-vs-One strategy which splits a multi-class classification into one binary classification problem per each pair of classes.

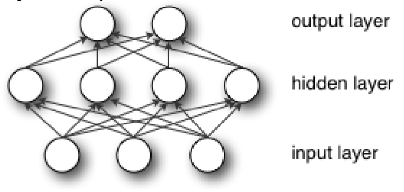
2) Algorithm implementation: The `sklearn` library provides the implementation of Support Vector Machine classifier, i.e SVC. And we can easily change the kernel function like `linear` and `rbf`.

Listing 3. SVC
```
from sklearn import svm
from sklearn.metrics import
    accuracy_score,f1_score

model = svm.SVC(kernel='linear')
model.fit(X_train,y_train)

y_pred = model.predict(X_test)
print("The␣model␣accuracy␣is:",
    accuracy_score(y_test,y_pred),
'f1_score␣is:',f1_score(y_test,y_pred,
    average='macro'))
```

D. MultiLayer Perceptron Neural Network(MLPNN)

1) Algorithm principle: A Multi-Layer Perceptron(MLP) can be viewed as a logistic regression classifier where the input is first transformed using a learnt non-linear transformation Φ. This transformation projects the input data into a space where it becomes linearly separable. This intermediate layer is referred to as a hidden layer. A single hidden layer is sufficient to make MLPs a universal approximator. [1]

Fig. 4. Multi-Layer Perceptron



For every hidden layer node z_i, there is a linear combination w of the features from the input layer x_i and a non-linear activation σ function to introduce complex representations. It is the same as in output layer y_i.

$$z_i = \sigma(net_i^{(1)}) \quad = \sigma(\sum_j w_{ij}^{(1)} x_j + b_i^1)$$
$$y_i = \sigma(net_i^{(2)}) \quad = \sigma(\sum_j w_{ij}^{(2)} z_j + b_i^2)$$

Then, we calculate the loss function, the distance from predition $\hat{y}$ and the ground truth y, like cross entropy loss between a probability distribution $\hat{y}$ and a class truth y.

$$L(\hat{y}, y) = -\log(\frac{exp(\hat{y}[y])}{\sum_j exp(\hat{y}[j])})$$

Except for cross entropy loss, multiclass classification can also use multi-margin loss.

$$L(\hat{y}, y) = \frac{\sum_i \max(0, \delta - \hat{y}[y] + \hat{y}[i])}{\text{num\_class}}$$

Finally, we perform backpropagation to use the loss gradient to update perceptron parameters, i.e computing the gradient for each weight and bias according to the chain rule. Take the cross entropy loss for example.

$$\begin{aligned}
\frac{\partial L}{\partial y_i} &= \frac{\exp(y_i)}{\sum_i \exp(y_i)} \\
\frac{\partial y_i}{\partial w_{ij}^2} &= \frac{\partial y_i}{\partial net_i^{(2)}} \cdot \frac{\partial net_i^{(2)}}{\partial w_{ij}^2} \\
&= \sigma'(net_i^{(2)}) z_j \\
\frac{\partial L}{\partial w_{ij}^2} &= \frac{\partial L}{\partial y_i} \cdot \frac{\partial y_i}{\partial w_{ij}^2} \\
&= \frac{\exp(y_i)}{\sum_i \exp(y_i)} \cdot \sigma'(net_i^{(2)}) z_j
\end{aligned}$$

When we update the parameters, some strategies are applied to get faster trainning speed and better coverge point. Adam is one of the best optimizer to do this.

Finally, after several times of traversing all dataset, i.e epochs, the loss of the training set and validation set stagnated at a lower point. And we take the parameter at the time to predict.

2) Algorithm implementation: We use Pytorch deep learning framework to build our model. There are several losses and optimizers to choose from. We take cross entropy loss and multi-margin loss and pass them to the model to get different results. And Adam is applied as the optimizer.

The implementation follows the Pytorch lightning template[2].

Listing 4. MultiLayer Perceptron
```
class LitClassifier(pl.LightningModule):
  def __init__(self, loss, hidden_dim
      =128, learning_rate=1e-3):
    super().__init__()
    self.save_hyperparameters()

    self.l1 = torch.nn.Linear(28 * 28,
        self.hparams.hidden_dim)
    self.l2 = torch.nn.Linear(self.
        hparams.hidden_dim, 10)

  def forward(self, x):
    x = x.view(x.size(0), -1)
```

[2]https://github.com/PyTorchLightning/deep-learning-project-template/blob/master/project/lit_mnist.py

```
        x = torch.relu(self.l1(x))
        x = torch.relu(self.l2(x))
        return x

    def training_step(self, batch,
        batch_idx):
        x, y = batch
        y_hat = self(x)
        loss = self.hparams.loss(y_hat, y)
        return loss

    def validation_step(self, batch,
        batch_idx):
        x, y = batch
        y_hat = self(x)
        loss = self.hparams.loss(y_hat, y)
        y_hat = torch.argmax(y_hat, dim=1)
        acc = pl.metrics.functional.
            classification.accuracy(y_hat,y)
        self.log_dict({'val_loss': loss,'
            val_accuracy': acc})

    def test_step(self, batch, batch_idx):
        x, y = batch
        y_hat = self(x)
        loss = self.hparams.loss(y_hat, y)
        y_hat = torch.argmax(y_hat, dim=1)
        f1 = pl.metrics.functional.
            classification.f1_score(y_hat,y)
        acc = pl.metrics.functional.
            classification.accuracy(y_hat,y)
        self.log_dict({'test_loss': loss,'
            f1_score':f1 ,'accuracy': acc})

    def configure_optimizers(self):
        return torch.optim.Adam(self.
            parameters(), lr=self.hparams.
            learning_rate)
```

E. Convolutional Neural Networks

1) Algorithm principle: Convolutional networks, also known as convolutional neural networks, or CNNs, are a specialized kind of neural network for processing data that has a known grid-like topology. [2]

In its most general form, convolution is an operation on two functions of a real-valued argument. One is the input $x(t)$, and the other is the weight $w(a)$. If we apply such a weighted average operation at every position of x, we obtain a new function s providing a smoothed estimate of the x.

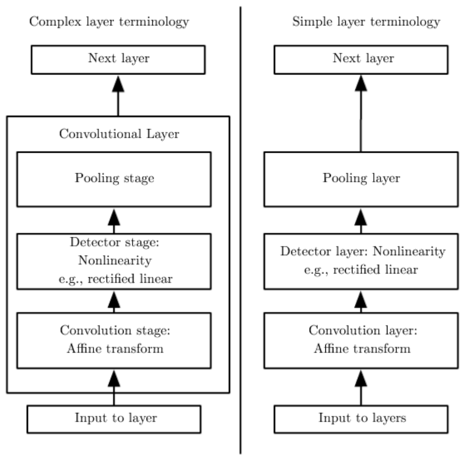
$$s(t) = (x * w)(t) = \int x(a)w(t-a)da$$

This operation is called convolution. In convolutional network terminology, w is referred as the kernel and s as the feature map.

Convolution leverages three important ideas that can help improve a machine learning system:sparse interactions, parameter sharing and equivariant representations.

Unlike dense connected layers in Multi-layer Perceptron Neural Networks, the convolutional layers only have few parameters shared by all output nodes, and one input only interacts with near inputs. That greatly reduced the amount of parameters and introduce the prior of local correlation.

Fig. 5. Convolutional Layer


Complex layer terminology
Next layer
Convolutional Layer
Pooling stage
Detector stage: Nonlinearity e.g., rectified linear
Convolution stage: Affine transform
Input to layer
Simple layer terminology
Next layer
Pooling layer
Detector layer: Nonlinearity e.g., rectified linear
Convolution layer: Affine transform
Input to layers

A typical layer of a convolutional network consists of three stages .In the first stage, the layer performs several convolutions in parallel to produce a set of linear activations.

$$\text{out} = \text{bias} + \sum \text{weight} * \text{input}$$

In the second stage, each linear activation is run through a nonlinear activation function, such as the rectified linear activation function. This stage is sometimes called the detector stage.

$$\text{ReLU}(x) = (x)^+ = \max(0, x)$$

In the third stage, we use a pooling function to modify the output of the layer further.

A pooling function replaces the output of the net at a certain location with a summary statistic of the nearby outputs.For example, the max pooling operation reports the maximum output within a rectangular neighborhood.

In all cases, pooling helps to make the representation approximately invariant to small translations of the input. It can vary the layers' size and is proved to be very useful when attracting features.

2) Algorithm implementation: We applied the deep learning library `pytorch` to implement this algorithm. We chose the LeNet network mentioned in [2]. And it was proven to be effective on the MNIST dataset.

LeNet is composed of 2 convolutional layers and 2 fully-connected layers. The output then goes through Softmax function and the loss is cross-entropy loss.

Listing 5. LeNet

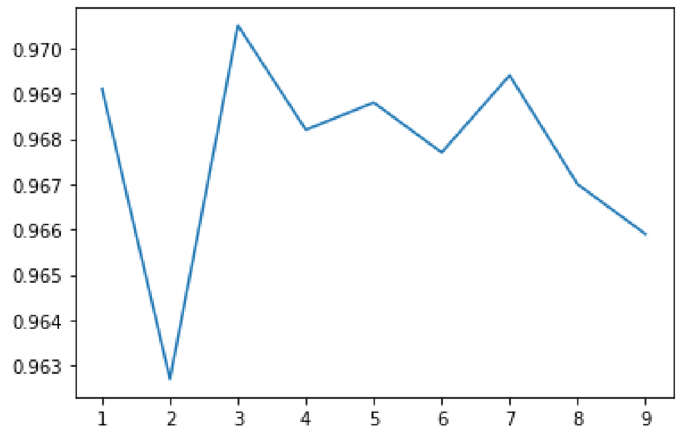
```
class LeNet(nn.Module):
    def __init__(self):
        super(LeNet, self).__init__()
        self.conv1 = nn.Conv2d(1, 6, 3,
            1)
        self.conv2 = nn.Conv2d(6, 16, 3,
            1)
        self.fc1 = nn.Linear(400, 128)
        self.fc2 = nn.Linear(128, 10)

    def forward(self, x):
        x = self.conv1(x)
        x = torch.tanh(x)
        x = F.avg_pool2d(x, 2)
        x = self.conv2(x)
        x = torch.tanh(x)
        x = F.avg_pool2d(x, 2)
        x = torch.flatten(x, 1)
        x = self.fc1(x)
        x = self.fc2(x)
        output = F.log_softmax(x, dim=1)
        return output
```

V. EXPERIMENTAL RESULTS AND ANALYSIS

A. K-Nearest Neighbor

Now we get the model and the data,then we can start training and testing.We choosed different k for knn algorithm and run the prediction.The following graph shows the variation of accuracy between different k from 1 to 9.

Fig. 6. Selection of k



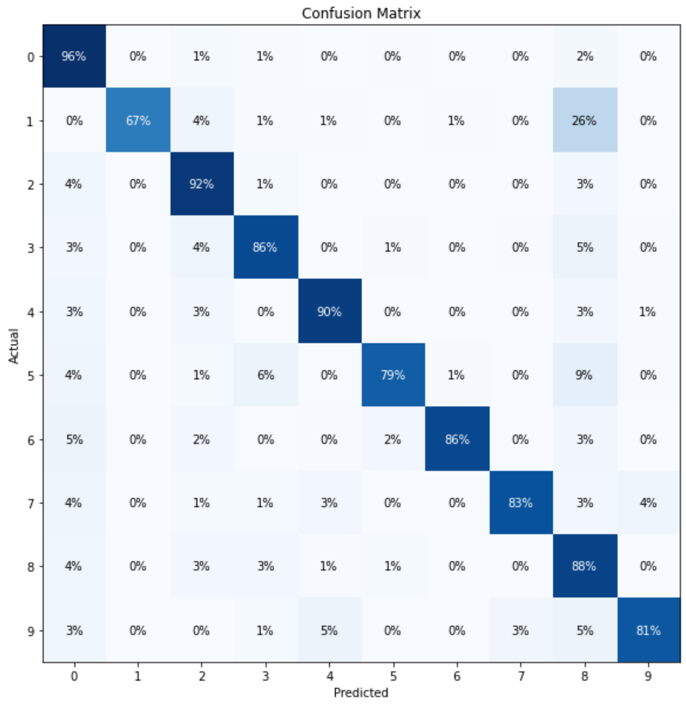
We can get from the graph that it's not true that the bigger the k, the better the result. Among all the values of k, 3 get the best result and from 7 to 9 the accuracy is dropping as the k increases. Finally, the best result is the same as K-nearest-neighbors, Euclidean (L2) model with 3.09% error rate from Kenneth Wilder, U. Chicago. [3]

B. Bayesian Decision

Now we get the model and the data,then we can start training and testing. We finally choosed *smooth* = 1000 in 0-255 gray scale space and get 0.815 on the monovariable model, and 0.8459 on the multivariable model.

Moreover, we can look into accuracies in every digit class and plot the confusion matrix. The following is the confusion matrix of the result the multivariable model produce. Comparing to the average image in 3, we can find that the zero(0) is far different from other digit and get the highest accuracy but the distribution of one(1) is so similar to that of the eight(8) that many ones were mistaken as eight. The result matched our expectations.

Fig. 7. Confusion Matrix



Actual \ Predicted	0	1	2	3	4	5	6	7	8	9
0	96%	0%	1%	1%	0%	0%	0%	0%	2%	0%
1	0%	67%	4%	1%	1%	0%	1%	0%	26%	0%
2	4%	0%	92%	1%	0%	0%	0%	0%	3%	0%
3	3%	0%	4%	86%	0%	1%	0%	0%	5%	0%
4	3%	0%	3%	0%	90%	0%	0%	0%	3%	1%
5	4%	0%	1%	6%	0%	79%	1%	0%	9%	0%
6	5%	0%	2%	0%	0%	2%	86%	0%	3%	0%
7	4%	0%	1%	1%	3%	0%	0%	83%	3%	4%
8	4%	0%	3%	3%	1%	1%	0%	0%	88%	0%
9	3%	0%	0%	1%	5%	0%	0%	3%	5%	81%

C. SVM and MLP Analysis

We train different models on the same training set and test on the same test set but the sizes of training set differ because of too long training time for SVM models.

SVM need to solve optimization problem with precise method and one-vs-one strategy requires many SVM models built for every pair of classes. That cause huge repeated float calculation on CPU. Especially with a non-linear kernel like rbf, the training process lasts longer. So part of the training set is applied instead of the whole.

TABLE I
COMPARISON OF SVM WITH DIFFERENT SIZES OF TRAING DATA

Model	linear(10000)	linear(20000)	linear(30000)	rbf(30000)
Time(s)	11.2s	37.49	80.31	103.42
Accuracy	0.9172	0.9257	0.9348	0.9742
F1-score	0.9158	0.9245	0.9337	0.9740

For multilayer perceptron models, if the training set is too small, the model won't behave fairly well. Therefore we trained the perceptron models on the whole training set. As the MLP model requires sequential learning, i.e gradient descent, so it may take longer time to train and tune. We only trained 10 epochs on the whole training set accelerated by GPU.

We can learn from the table that for SVM model, rbf kernel have better representation of the data than linear model. With

Fig. 8. Multiple Perceptron Validation Accuracy when training

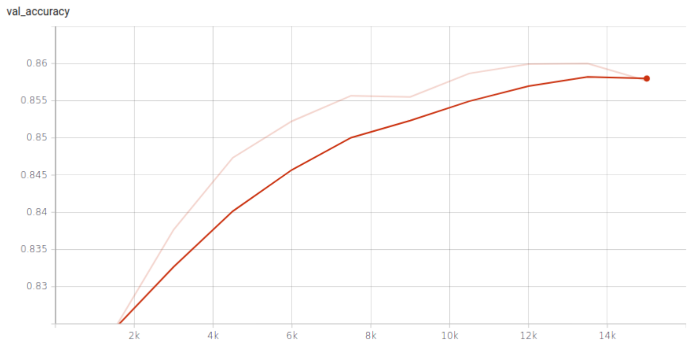
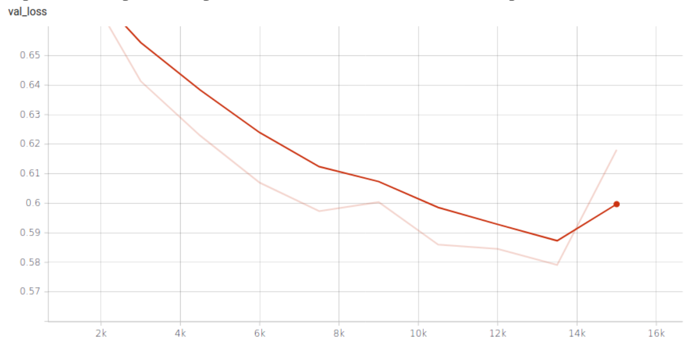


Fig. 9. Multiple Perceptron Validation Loss when training



small data set, SVM performs better and runs faster with even less data.

For multilayer perceptron models, the multi-margin loss have weak constraint for the classification, while cross entropy loss perform better.

TABLE II
COMPARISON OF ACCURACY OF DIFFERENT MODELS

Model	Classification	Training Set	Accuracy	F1-score
SVM(linear)	one-vs-one	30000	0.9348	0.9337
SVM(rbf)	one-vs-one	30000	0.9742	0.9740
MLP(ce)	multiclass	60000	0.8421	0.8417
MLP(margin)	multiclass	60000	0.6244	0.6230

D. Convolutional Neural Networks

We train the model for 10 epochs optimized with Adadelta and learning rate 0.5. The accuracy and loss value thoughout training is at below. We finally achieved 98.3% accuracy on the test set.

CNN show its impressive ability to classify the image. With only 1 epoch, the model achieved 95% accuracy on validation set. It outperforms other linear model like SVM and Perceptron with speed and result.

E. Comparison

We compared the models mentioned on time consumption, precision and recall performances, in table TABLE III and in figures Fig.12. We can learning several facts from the comparison.

Fig. 10. Accuracy on validation set

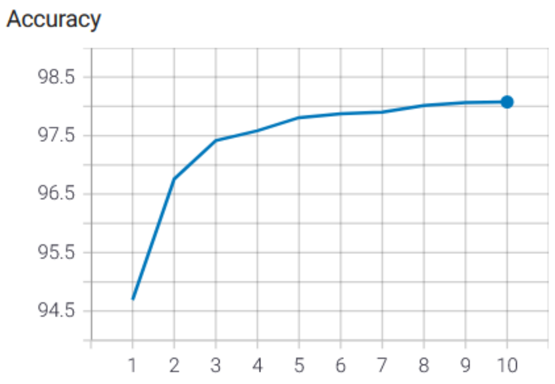
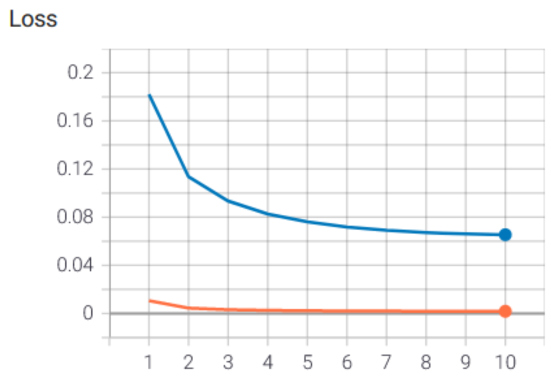


Fig. 11. Loss on training set and validation set

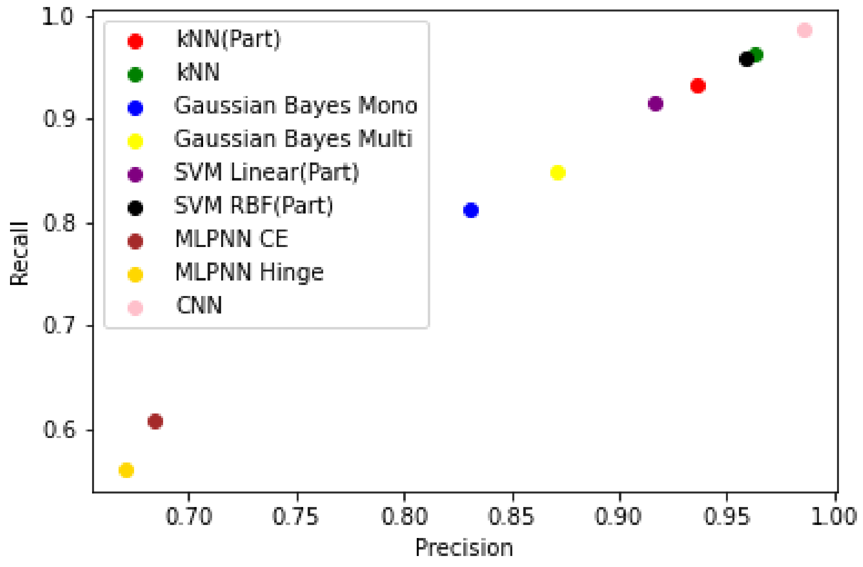


- Applying the whole training set was wasting time for simple models like kNN and SVM, as they perform excellently even with 1/6 of the training set.
- Non-linear classifiers usually have better performance, such as RBF Kernel vs. Linear Kernel in SVM and Cross Entropy vs. Hinge Loss in MLPNN.
- The metrics of MLPNN is unexpectedly bad. It's probably due to the weak ability of two-layer perceptron.
- CNN has the highest f1 score and achieve almost 100% in precision and recall. That means CNN's local pixel prior is effective.

REFERENCES

[1] Christopher M Bishop. *Pattern recognition and machine learning*. springer, 2006.
[2] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016.
[3] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.
[4] Z.H. Zhou. *Machine Learning*. Tsinghua University Press, 2016.

Fig. 12. Comparison of different models



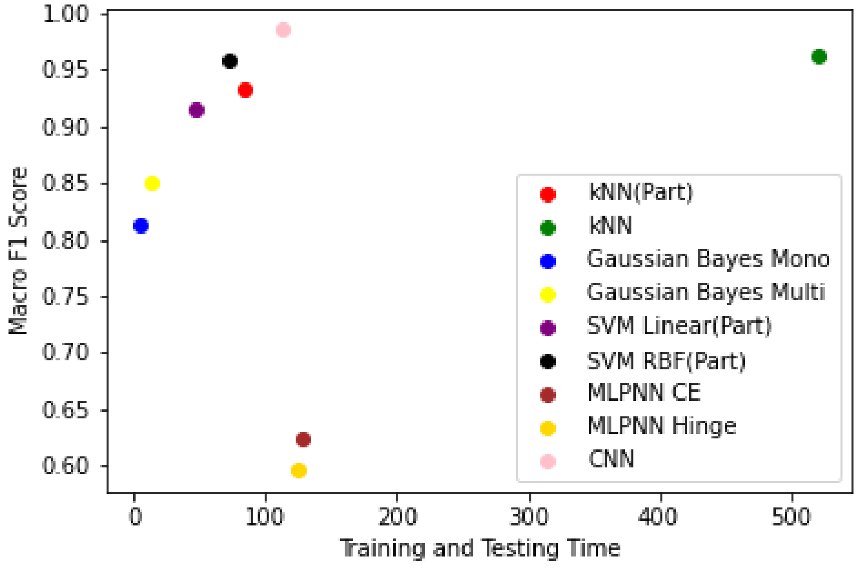


TABLE III
COMPARISON OF DIFFERENT MODELS

No.	Model Type	Description	Training set	Training Time(s)	Testing Time(s)	Precision	Recall	F1-macro
1	K-Nearest Neighbor	k=2	10000	0.01	84.64	0.9358	0.9324	0.9327
2	K-Nearest Neighbor	k=2	60000	0.00	520.16	0.9635	0.9621	0.9623
3	Bayesian	Gaussian Mono-variable	60000	1.12	3.76	0.8306	0.8114	0.8133
4	Bayesian	Gaussian Multi-variable	60000	1.06	12.54	0.8706	0.8478	0.8498
5	SVM	Linear Kernel	10000	17.50	28.69	0.9166	0.9157	0.9158
6	SVM	RBF Kernel	10000	29.18	43.16	0.9593	0.9590	0.9590
7	MLPNN	epoch=10 Cross Entropy Loss	60000	128.26(GPU)	0.04	0.6839	0.6077	0.6250
8	MLPNN	epoch=10 Multi-margin Loss	60000	124.02(GPU)	0.04	0.6708	0.5602	0.5965
9	CNN	epoch=10 LeNet	60000	112.50(GPU)	0.03	0.9853	0.9854	0.9853