CS 6241 Final Project Report: Deep Kernel Learning for Satellite Imagery Classification

Aneesh Heintz

1 Introduction

Model prediction uncertainty is a crucial metric for various types of regression and classification tasks; especially in applications such as autonomous systems and medical diagnoses. Deep neural networks are flexible parametric models that can fit complex nonlinear patterns in data. Convolutional neural networks (CNN) are powerful pattern-recognition tools that have proven to be state of the art for image classification tasks. However, as they are increasingly implemented, their associated uncertainty estimates are not very accurate and do not produce confidence or uncertainty bounds on the predictions they create.

However, Bayesian inference has proven to be successful for learning under uncertainty. But the large number of parameters in CNNs means Bayesian inference is difficult to implement in the context of CNNs. Gaussian Processes (GP) are a powerful non-parametric tool in machine learning and allow predictions about data to be made by incorporating prior knowledge. GPs are easier suited for Bayesian inference. For a given training set of data, there are potentially infinitely many functions that fit the data. GPs assign probabilities to each function and the mean of the resulting probability distribution is the most probable characterization of the data. More importantly, GPs produce uncertainty estimates.

Using a deep neural network and Gaussian Process in sequence is known as deep kernel learning (DKL). DKL emerged as a useful research field to address the individual problems associated with neural networks and GPs [0]. It leverages the flexibility and interpretability uncertainty estimation framework of GPs with the ability to learn high-dimensional functions inherent in deep neural networks. Thus the neural network learns the kernel operator of a GP, which is in turn used to perform inference tasks [0].

This paper presents the creation and training of a convolutional deep kernel learning model to create classification predictions with associated uncertainty estimates. Therefore, the resulting model would reach performances similar to that of a CNN while adding a confidence metric of its prediction. Experiments are conducted on both synthetic and real datasets.

2 Related work

2.1 Neural Network Uncertainty

There are several methods that predate deep kernel learning that involve adding uncertainty bounds to predictions using GPs and CNNs. [1] shows that the output of a residual CNN with a 2D convolutional network prior over the weights and biases is a Gaussian Process in the limit of infinitely many convolutional filters. They show that state of the art and practical architectures such as CNNs and ResNets have equivalent GP representations. If each hidden layer has an infinite number of convolutional filters, the network prior is equivalent to a GP. [2] addresses uncertainty bounds of predictions using model dropout during training. They proposed a new theoretical framework that casts dropout training in the learning process of neural networks as Bayesian inference. This allows uncertainty estimates to be created directly from existing models.

2.2 Deep Kernel Learning

Deep Kernel learning was first proposed by [5] in an effort to combine the most useful assets associated with neural networks and GPs. GPs were popular because it is proven that bayesian neural nets with an infinite width converge to GPs with a particular kernel function. They are also flexible and interpretable. Neural networks, however, are superior in understanding representations in high-dimensional data. Therefore [5] trains a neural network and uses the top-level features of the deep neural network model as inputs to a GP. They show that DKL obtains results better than if its respective neural network model were trained on its own. The proposed DKL methods by [5] applied only to single-output regression problems and prohibits stochastic training. [6] expands upon [5] by proposing a new deep kernel learning model, Stochastic Variational DKL, that enables stochastic training, multi-task and multi-output learning. The SV-DKL architecture achieves competitive performance to stand-alone neural network architectures and beats the performance of similar methods that use neural networks and GPs in a coherent model.

3 Deep kernel learning method

The dataset being used contains n input images, $X = x_1, ..., x_n$. Each image is described by a two-dimensional matrix with shape $m \times m$. Each image has an associated class category, y_i. The deep kernel learning applies an arbitrary convolutional neural network, $h(x, w)$, on the input images and is parameterized by weights w. The convolutional neural network creates embeddings, $h_i(w, x)$ that are used as input features to a GP. For notation purposes, let the base kernel of a GP framework is defined as $k(x_i, x_j|\theta)$, where θ is the parameters of the base kernel. Under the condition where the non-linear transformation, $h(x, w)$ given by the neural network acts on the inputs of the entire model, the base kernel becomes $k(w, h(x_i, w), h(x_j, w)|\theta)$. The final layer of the convolutional neural network is passed as input to a Gaussian process:

$$f(h(x, w)) \sim \mathcal{GP}(\mu, k_\gamma)$$

where the mean vector, $\mu_i = \mu(x_i)$, and covariance matrix, $(K_{h,h})_{ij} = k_\gamma(h(x_i, w), h(x_j, w))$, is determined from the mean function and covariance kernel of the $f(h(x, w))$.

3.1 Kernel (Covariance) Function

Then any collection of function values f has a joint Gaussian distribution,

$$f(h(x, w)) = [f(h_1), ..., f(h_n)]^T \sim \mathcal{N}(\mu, K_{h,h})$$

The deep kernel learning model consists of a convolutional neural network followed by a Gaussian Process layer with as many outputs as there are classes in the dataset. The Gaussian Process layer uses an RBF kernel function:

$$k_{RBF}(x, x') = cov(f(x), f(x')) = a^2 exp(-\frac{||x - x'||^2}{2l^2})$$

where a and l are kernel hyper-parameters that controls the amplitudes and frequencies of the GP functions. The RBF kernel holds the assumption that function values at nearby inputs are more correlated than function values at far away inputs.

3.2 Loss Function

The model is trained according to the following Variational Evidence Lower Bound (ELBO) loss function [7].

$$\mathcal{L}_{\text{ELBO}} = \mathbb{E}_{p_{\text{data}}(y,\mathbf{x})}\left[\mathbb{E}_{p(f|\mathbf{u},\mathbf{x})q(\mathbf{u})}\left[\log p(y \mid f)\right]\right] - \beta\, \text{KL}\left[q(\mathbf{u}) \| p(\mathbf{u})\right]$$

Here, $q(u)$ is the variational distributions fo the inducing function values, and $p(u)$ is the prior distribution for the inducing function values. The loss function was implemented as a maximum likelihood loss in gpytorch [7].

3.3 Training

The model is trained in an end-to-end manner optimized by SGD according to the variational Evidence Lower Bound loss. Learning rates and training duration varies based on the dataset the method is applied to. The model is written on the pytorch and gpytorch (developed at Cornell!) [7] frameworks and run on a Tesla K80 GPU using Google Colab.

4 Intermediate Goal: Cropped-region MNIST Dataset

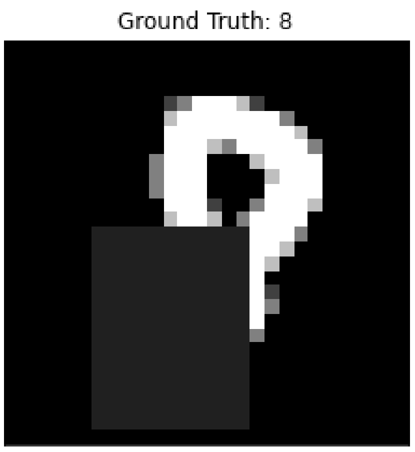
4.1 Dataset relevance

The synthetic dataset used is the MNIST dataset [3] with regions of each image randomly cropped out. It was selected as an intermediate goal to demonstrate progress and that the overall method worked correctly in a ”controlled environment.” The dataset is provided through torchvision as separate training and validation sets. In total, 60 thousand training and 10 thousand validation samples are in each respective set.

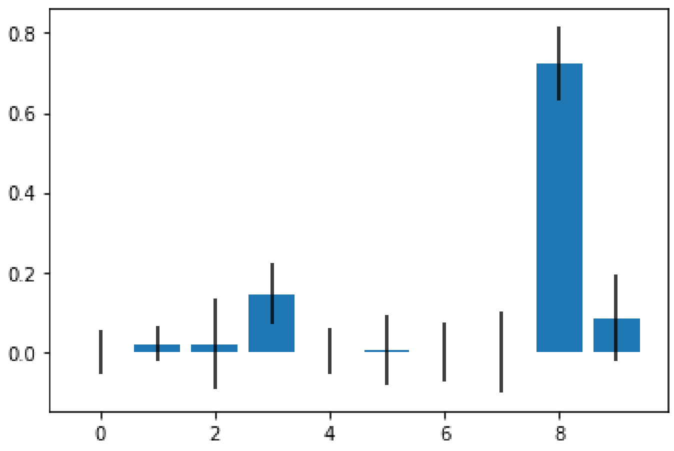
4.2 Analysis

The architecture of the CNN used in the DKL model for training on the MNIST dataset is defined by a simple encoder. The DKL method is trained in an end-to-end manner and use a convolutional neural network in succession with a GP to obtain uncertainty estimates without decreasing performance. A GP uses the weights of the CNN's final layer, before the softmax activation function, as inputs. The DKL model creates a confidence interval for each class prediction. The model achieves a 94.5% accuracy when categorizing the dataset on previously unseen cropped handwriting samples.

Figure 1 demonstrates one sample prediction by the model. Figure 1(a) is the input handwriting sample with a randomly cropped region. It corresponds to a ground truth value of 8. Figure 1(b) shows the distribution of the DKL model's GP for the respective handwriting sample. The prediction output shows a prediction value along with the variance scores for each class.



(a) Handwriting sample input



(b) Prediction distribution of input shown in 1(a)

Figure 1: Sample prediction of the DKL model trained on cropped-region MNIST

5 UC Merced Land Use Dataset

The UC Merced Land Use dataset [4] is a collection of satellite images of different categories that appear in urban areas. The dataset includes images extracted from large images from the USGS National Map Urban Area Imagery collection fro various urban areas around the United States. The consists of 21 different classes, such as agriculture, beaches, harbors, runways, golf courses, etc. Each class is associated with 100 images of size 256×256 pixels. Each pixel corresponds to 1 foot. Although this real-world dataset does not encompass all types of land regions that can be seen in satellite imagery, it nonetheless provides a good basis to demonstrate the proposed method.

5.1 Dataset relevance

Satellite imagery is useful for many reasons. Obviously, it is useful to gather information about the weather. Satellite imagery can aid tracking the spread of wildfires or the progression of hurricanes. Satellite imagery can also be useful for gathering economic information, such as measuring ship or airplane movements around the world. Measuring how busy each port or airport is can be quite informative and lucrative. This is especially important with regard to the current impacts associated with COVID-19. Satellite imagery has demonstrated the impact of the virus around the world. In Venice, for example, it can be seen through satellite imagery that the canals have cleared up and are far less polluted now that boats and ships aren't operating in the nearby waters. Lastly, satellite imagery is often used for surveillance purposes. Both the government and human rights organizations use imagery of urban and rural areas to track vehicle movements and other activity at known locations over time.

The useful applications of satellite imagery are very broad. However, much of the aforementioned use cases for satellite imagery is analyzed manually. Automating classification, identification, and segmentation tasks on satellite imagery through methods like the one proposed to extract and analyze useful information could yield impressive results faster. When automated, such processes must have associated uncertainty estimates to be used in future decision making tasks.

5.2 Experiments

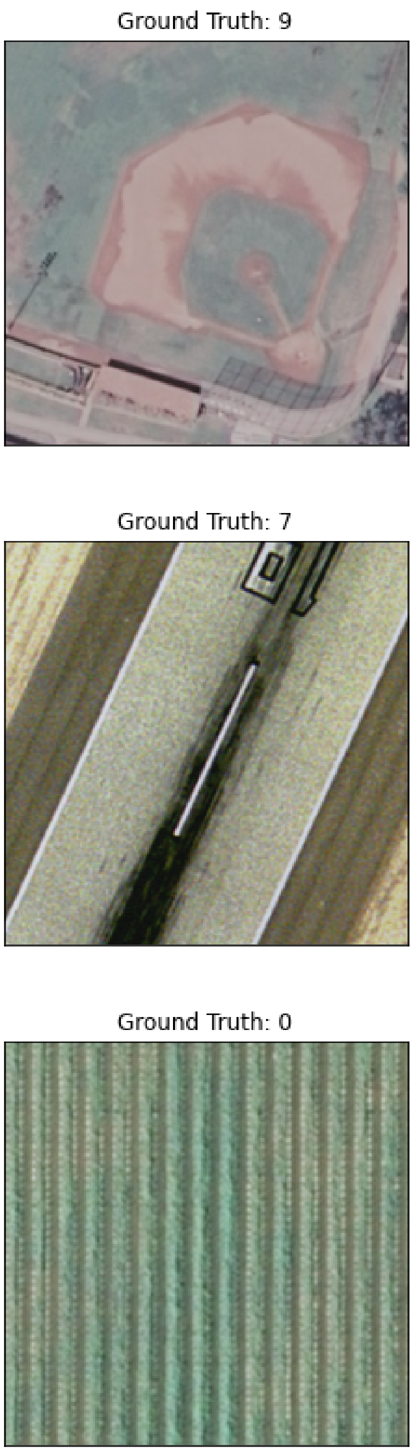
5.2.1 Performance comparison to a Stand-alone CNN

Using a simple CNN encoder, the DKL method achieves a performance accuracy of 75% on the UC Merced Land Use dataset. While the accuracy of the model is not as high as when it was applied to the synthetic MNIST dataset, it outperforms the same CNN encoder architecture trained end-to-end but without a GP. The comparative results are summarized in Table 1.

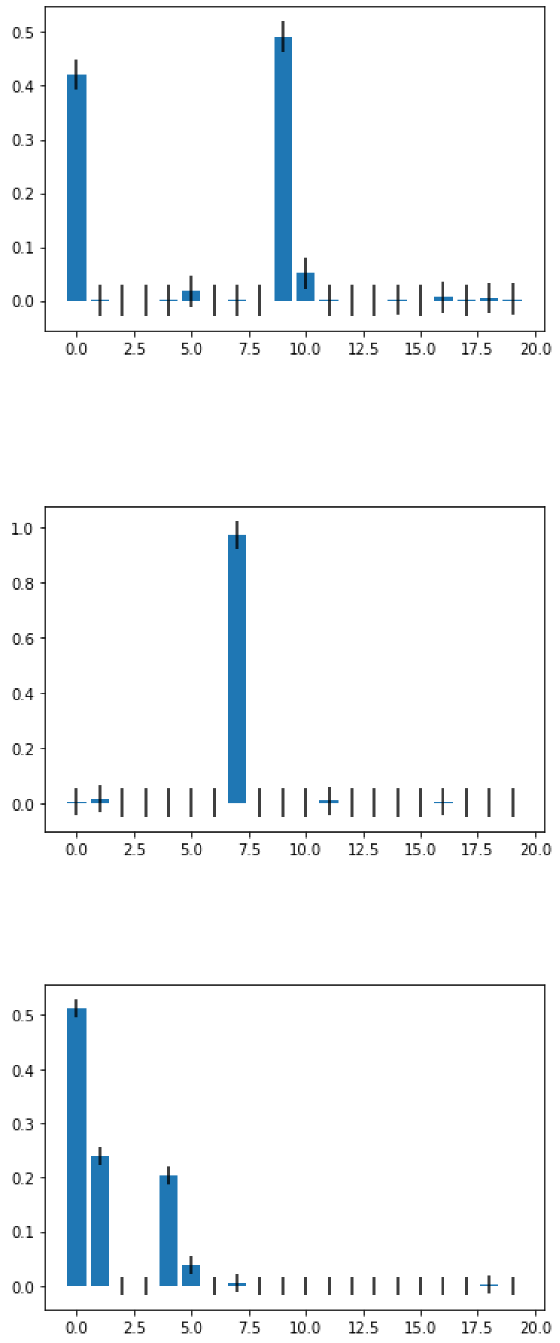
Table 1: Comparison between the DKL CNN-GP and a stand-alone CNN.

	Validation Loss	**Validation Accuracy**
DKL CNN-GP	1.18	75.20%
DKL CNN-GP (noisy test data)	1.28	70.99%
Stand-alone CNN	2.36	69.21%

An additional benefit of using the DKL CNN-GP over the stand-alone CNN is the confidence metrics associated with the predictions. For demonstration, Figure 2 shows a few selected example prediction distributions with respective confidence intervals from the DKL CNN-GP with a given satellite input image.



(a) Sample input



(b) Prediction distribution of input shown in 1(a)

Figure 2: Sample predictions of the DKL model trained on the UC Merced Land Use dataset

5.2.2 Adding Noise

To demonstrate the usefulness of the confidence intervals associated with class predictions, a study was conducted on how uncertainty estimates varied with data that has a distribution that differs from the underlying distribution of the training data. As the UC Merced Land Use dataset has already been pre-processed to filter out noise, shot (Poisson) noise was added back in to each image in a validation set to simulate the effect that electronics on board the satellite would have on the camera during operation. This mimics an operational environment where the proposed method would be used on-board a spacecraft, something not currently done. On this dataset, the DKL method achieves a performance accuracy of 70.99%. Comparisons with the aforementioned experiments is detailed in Table 1. Selected prediction distributions are shown in Figure 3. They show an increase in the variance of the predictions, as expected when noise is added to images. However, it demonstrates that the proposed DKL CNN-GP method, when trained end-to-end, is able to achieve similarly competitive performances on a dataset that has a different underlying distribution from the training data.

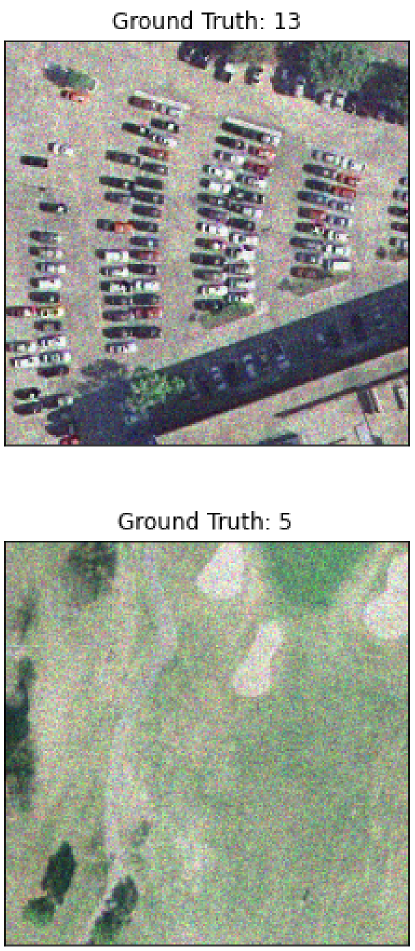
5.2.3 Comparison to a Pre-trained CNN-GP

[6] demonstrated that training deep kernel learning methods end-to-end produced better results than training a GP with inputs derived from a pre-trained CNN. The last experiment performed further validated this conclusion. This experiment trained the DKL method in the same manner as the ones before. However, the CNN component of the DKL model was a ResNet18 module pre-trained on ImageNet. After training, the model acheived an accuracy of roughly 67%, similar to that of the stand-alone CNN.

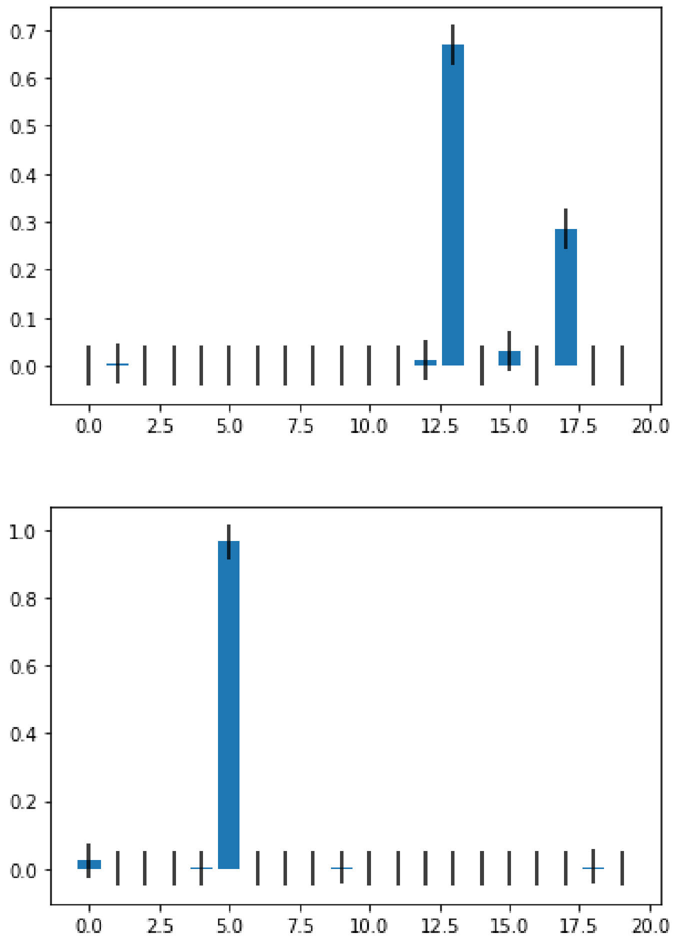
6 Conclusion

In the robotics and space technology community, deep learning is a widely studied field. The impacts of implementing learning algorithms is wide. However, their implementations are not yet trustworthy, robust, and reliable due to the ”black box” nature of neural networks and the lack of proper guidelines to properly and successfully train models. However, adding confidence metrics on deep learning predictions is a step in the right direction.

On both the synthetic MNIST dataset and the real-world Land Use dataset, the GP trained on the final layer of a simple encoder in an end-to-end fashion acheives a performance competitive to a simple stand-alone CNN. Additionally, this paper demonstrated that the end-to-end



(a) Sample input



(b) Prediction distribution of input shown in 1(a)

Figure 3: Results of shot noise added to the UC Merced Land Use dataset

training of the proposed DKL CNN-GP method is able to understand data with a different underlying distribution than what the model was trained on. The GP is also able to create confidence intervals associated with resulting class predictions from the neural network's output weights. Confidence metrics are extremely powerful and informative. They can aid in a wide variety of tasks, such as autonomous decision making and streamlined analysis of data.

7 References

[0] Dasgupta, Sambarta & Sricharan, Kumar & Srivastava, Ashok. Finite Rank Deep Kernel Learning. NeurIPS, 2018. http://bayesiandeeplearning.org/2018/papers/98.pdf

[1] Garriga-Alonso, Adrià & Aitchison, Laurence & Rasmussen, Carl. Deep Convolutional Networks as shallow Gaussian Processes. ICLR, 2019.

[2] Gal, Yarin & Ghahramani, Zoubin. (2015). Dropout as a Bayesian Approximation: Representing Model Uncertainty in Deep Learning. Proceedings of The 33rd International Conference on Machine Learning.

[3] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. "Gradient-based learning applied to document recognition." Proceedings of the IEEE, 86(11):2278-2324, November 1998.

[4] Yi Yang and Shawn Newsam, "Bag-Of-Visual-Words and Spatial Extensions for Land-Use Classification," ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems (ACM GIS), 2010.

[5] Andrew Gordon Wilson, Zhiting Hu, Ruslan Salakhutdinov, and Eric P Xing. 2016. Deep

kernel learning. In Proceedings of the 19th International Conference on Artificial Intelligence and Statistics. 370–378.

[6] Andrew G Wilson, Zhiting Hu, Ruslan R Salakhutdinov, and Eric P Xing. Stochastic variational deep kernel learning. In Advances in Neural Information Processing Systems, pp. 2586–2594, 2016a.

[7] Gardner, Jacob R., Geoff Pleiss, David Bindel, Kilian Q. Weinberger, and Andrew Gordon Wilson. "GPyTorch: Blackbox Matrix-Matrix Gaussian Process Inference with GPU Acceleration." In Advances in Neural Information Processing Systems (2018)

9 Appendix: MNIST Code

```
! pip install gpytorch
import torch
import torchvision
from torchvision import datasets, transforms
import matplotlib.pyplot as plt
import torch.nn as nn
import torch.nn.functional as F
from torch.optim import SGD, Adam
import torch.optim as optim
from torch.optim.lr_scheduler import StepLR
import tqdm
from torch.optim.lr_scheduler import MultiStepLR
import gpytorch
import math
import numpy as np

use_cuda = torch.cuda.is_available()
device = torch.device("cuda" if use_cuda else "cpu")
num_classes = 10
kwargs = {'num_workers': 1, 'pin_memory': True} if use_cuda else {}
args = {'log_interval': 1, 'learning_rate': 1e-1, 'step_size': 75, 'gamma': 0.1, 'n_epochs':
     15, 'batch_size_train': 128, 'batch_size_test': 64}
torch.manual_seed(1)
print(use_cuda)

normalize = torchvision.transforms.Normalize((0.1307,), (0.3081,))
train_loader = torch.utils.data.DataLoader(
  torchvision.datasets.MNIST('/files/', train=True, download=True,
                             transform=torchvision.transforms.Compose([
                               torchvision.transforms.ToTensor(),
                               normalize,
                                 transforms.RandomErasing()
                             ])),
  batch_size=args['batch_size_train'], shuffle=True)
```

```python

test_loader = torch.utils.data.DataLoader(
  torchvision.datasets.MNIST('/files/', train=False, download=True,
                             transform=torchvision.transforms.Compose([
                               torchvision.transforms.ToTensor(),
                               normalize,
                               transforms.RandomErasing()
                             ])),
  batch_size=args['batch_size_test'], shuffle=True)

train_loader.dataset.train_data.shape, test_loader.dataset.train_data.shape

"""Run CNN on MNIST"""

class CNN(nn.Module):
    def __init__(self, num_features):
        super(CNN, self).__init__()
        self.conv1 = nn.Conv2d(1, 32, 3, 1)
        self.conv2 = nn.Conv2d(32, 64, 3, 1)
        self.dropout1 = nn.Dropout2d(0.25)
        self.dropout2 = nn.Dropout2d(0.5)
        self.fc1 = nn.Linear(9216, 128)
        self.fc2 = nn.Linear(128, num_features)

    def forward(self, x):
        x = self.conv1(x)
        x = F.relu(x)
        x = self.conv2(x)
        x = F.relu(x)
        x = F.max_pool2d(x, 2)
        x = self.dropout1(x)
        x = torch.flatten(x, 1)
        x = self.fc1(x)
        x = F.relu(x)
        x = self.dropout2(x)
        x = self.fc2(x)
        output = x
        return output

class GaussianProcessLayer(gpytorch.models.ApproximateGP):
    def __init__(self, num_dim, grid_bounds=(-10., 10.), grid_size=64):
        variational_distribution = gpytorch.variational.CholeskyVariationalDistribution(
            num_inducing_points=grid_size, batch_shape=torch.Size([num_dim])
        )
        variational_strategy = gpytorch.variational.MultitaskVariationalStrategy(
            gpytorch.variational.GridInterpolationVariationalStrategy(
                self, grid_size=grid_size, grid_bounds=[grid_bounds],
                variational_distribution=variational_distribution,
            ), num_tasks=num_dim,
        )
        super().__init__(variational_strategy)

        self.covar_module = gpytorch.kernels.ScaleKernel(
            gpytorch.kernels.RBFKernel(
                lengthscale_prior=gpytorch.priors.SmoothedBoxPrior(
                    math.exp(-1), math.exp(1), sigma=0.1, transform=torch.exp
                )
            )
        )
        self.mean_module = gpytorch.means.ConstantMean()
        self.grid_bounds = grid_bounds

    def forward(self, x):
        mean = self.mean_module(x)
        covar = self.covar_module(x)
        return gpytorch.distributions.MultivariateNormal(mean, covar)

class DKLModel(gpytorch.Module):
    def __init__(self, feature_extractor, num_dim, likelihood, grid_bounds=(-10., 10.)):
        super(DKLModel, self).__init__()
        self.feature_extractor = feature_extractor
        self.gp_layer = GaussianProcessLayer(num_dim=num_dim, grid_bounds=grid_bounds)
        self.grid_bounds = grid_bounds
        self.num_dim = num_dim
        self.likelihood = likelihood

    def forward(self, x):
        features = self.feature_extractor(x)
        features = gpytorch.utils.grid.scale_to_bounds(features, self.grid_bounds[0], self.
            grid_bounds[1])
        features = features.transpose(-1, -2).unsqueeze(-1)
        res_gp = self.gp_layer(features)
        return res_gp

def train(args, model, likelihood, mll, device, train_loader, optimizer, epoch):
    model.train()
    likelihood.train()

    total_loss = 0
    loss_fn = nn.NLLLoss()

    minibatch_iter = tqdm.notebook.tqdm(train_loader, desc=f"(Epoch {epoch}) Minibatch")
    with gpytorch.settings.num_likelihood_samples(8):
      for data, target in minibatch_iter:
          data, target = data.to(device), target.to(device)
          optimizer.zero_grad()
          output = model(data)
          loss = -mll(output, target)
          total_loss += loss.item()
          loss.backward()
          optimizer.step()
          minibatch_iter.set_postfix(loss=loss.item())
    total_loss /= len(train_loader.dataset)
    return total_loss

def test(model, likelihood, mll, device, test_loader):
    model.eval()
    likelihood.eval()
```

```python
    test_loss = 0
    correct = 0
    with torch.no_grad(), gpytorch.settings.num_likelihood_samples(16):
        for data, target in test_loader:
            data, target = data.to(device), target.to(device)
            output = model(data)
            loss = -mll(output, target)
            test_loss += loss.item()
            pred = likelihood(output).probs.mean(0).argmax(-1)
            correct += pred.eq(target.view_as(pred)).sum().item()

    test_loss /= len(test_loader.dataset)

    return test_loss, correct / len(test_loader.dataset)

num_dim = 100
NN = CNN(num_dim)
likelihood = gpytorch.likelihoods.SoftmaxLikelihood(num_features=num_dim, num_classes=num_
    classes)
model = DKLModel(NN, num_dim, likelihood)


model = model.to(device)
likelihood = likelihood.to(device)
optimizer = SGD([
    {'params': model.feature_extractor.parameters(), 'weight_decay': 1e-4},
    {'params': model.gp_layer.hyperparameters(), 'lr': args['learning_rate'] * 0.01},
    {'params': model.gp_layer.variational_parameters()},
    {'params': model.likelihood.parameters()},
], lr=args['learning_rate'], momentum=0.9, nesterov=True, weight_decay=0)
scheduler = StepLR(optimizer, step_size=args['step_size'], gamma=args['gamma'])

mll = gpytorch.mlls.VariationalELBO(likelihood, model.gp_layer, num_data=len(train_loader))

for epoch in range(1, args['n_epochs'] + 1):
  train_loss = train(args, model, likelihood, mll, device, train_loader, optimizer, epoch)
  test_loss, acc = test(model, likelihood, mll, device, test_loader)
  print('\n==> Epoch: {}, Train Loss: {:.4e}, Test Loss: {:.4e}, Test Acc: {:.4e}'.format(
      epoch, train_loss, test_loss, acc))
  scheduler.step()

"""Individual Testing"""

examples = enumerate(test_loader)

batch_idx, (example_data, example_targets) = next(examples)
example_data.shape
idx = 8
fig = plt.figure()
plt.tight_layout()
plt.imshow(example_data[idx][0], cmap='gray', interpolation='none')
plt.title("Ground Truth: {}".format(example_targets[idx]))
plt.xticks([])
plt.yticks([])
plt.show()

model.eval()
data, target = example_data.to(device), example_targets.to(device)
output = model(data)
observed_pred = likelihood(output)
preds = observed_pred.probs.mean(0).cpu()
pred = preds.argmax(-1)
pred_distribution = preds.cpu().detach().numpy()
var = observed_pred.probs.var(1).cpu().detach().numpy()
pred.eq(example_targets.view_as(pred)).cpu().sum().item() / float(len(pred))

pred, example_targets

pred[idx].item(), example_targets[idx].item()

plt.figure()
plt.bar(np.arange(10), pred_distribution[idx].squeeze(), yerr=var[idx].squeeze())
plt.show()

pred_distribution[pred[idx].item()]

var[idx]

preds[0]
```

10 Appendix: Stand-alone CNN Code

```
! pip install gpytorch
import torch
import torchvision
from torchvision import datasets, transforms
import torchvision.models as models
import matplotlib.pyplot as plt
import torch.nn as nn
import torch.nn.functional as F
from torch.optim import SGD, Adam
import torch.optim as optim
from torch.optim.lr_scheduler import StepLR
import tqdm
from torch.optim.lr_scheduler import MultiStepLR
import gpytorch
import math
import numpy as np
import os
from PIL import Image

# Commented out IPython magic to ensure Python compatibility.
from google.colab import drive
drive.mount('/content/drive/')
# % cd /content/drive/My Drive/Colab Notebooks/UCMerced_LandUse
```

```
use_cuda = torch.cuda.is_available()
device = torch.device("cuda" if use_cuda else "cpu")
kwargs = {'num_workers': 1, 'pin_memory': True} if use_cuda else {}
args = {'log_interval': 1, 'learning_rate': 1e-3, 'step_size': 50, 'gamma': 0.1, 'n_epochs':
     200, 'batch_size_train': 128, 'batch_size_test': 64}
torch.manual_seed(1)
print(use_cuda)

class Dataset(torch.utils.data.Dataset):
  def __init__(self, mode, split):
    path = './Images/'
    categories = os.listdir(path)
    imgs = []
    target = []
    for category_idx in (range(len(categories))):
      img_categories = os.listdir(path + categories[category_idx])
      for img_idx in (range(len(img_categories))):
        im_path = path + categories[category_idx] + '/' + img_categories[img_idx]
        imgs.append(im_path)
        target.append(category_idx)
    self.imgs = np.array(imgs)
    self.target = np.array(target)
    self.img_exceptions = [130, 183, 209, 243, 396, 504, 505, 506, 507, 622, 623, 624, 633,
                           770, 788, 858, 861, 863, 864, 865, 866, 867, 868, 869, 870, 915,
                           935, 945, 993, 1055, 1060, 1077, 1122, 1145, 1146, 1308, 1320,
                           1699, 1714, 1736, 1857, 2060, 2062, 2063]
    self.img_idxs = []
    for idx in range(len(imgs)):
      if idx not in self.img_exceptions:
        self.img_idxs.append(idx)

    self.img_idxs = np.array(self.img_idxs)
    np.random.shuffle(self.img_idxs)

    self.imgs = self.imgs[self.img_idxs]
    self.target = self.target[self.img_idxs]
    self.num_classes = max(self.target)
    self.categories = categories

    split_idx = int(len(self.imgs) * split)
    if mode == 'train':
        self.imgs = self.imgs[:split_idx]
        self.target = self.target[:split_idx]
    elif mode == 'test':
        self.imgs = self.imgs[split_idx:]
        self.target = self.target[split_idx:]

  def __len__(self):
    return len(self.imgs)

  def __getitem__(self, index):
    idx = index

    im_path = self.imgs[idx]
    im = Image.open(im_path)
    im = np.array(im) / 255.

    means = [0.5, 0.5, 0.5]
    stds = [0.5, 0.5, 0.5]

    for ch in range(3):
        im[:,:,ch] = (im[:,:,ch] - means[ch]) / stds[ch]

    return im.reshape((3,256,256)), self.target[idx]

train_dataset = Dataset('train', 0.7)
test_dataset = Dataset('test', 0.7)
train_loader = torch.utils.data.DataLoader(train_dataset, shuffle = True)
test_loader = torch.utils.data.DataLoader(test_dataset, shuffle = True)

class CNN(nn.Module):
    def __init__(self, num_features):
        super(CNN, self).__init__()
        self.conv1 = nn.Conv2d(3, 6, 5)
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(6, 16, 5)
        self.fc1 = nn.Linear(59536, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, num_features)

    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = x.view(-1, 59536)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        x = F.log_softmax(x)
        return x

class GaussianProcessLayer(gpytorch.models.ApproximateGP):
    def __init__(self, num_dim, grid_bounds=(-10., 10.), grid_size=64):
        variational_distribution = gpytorch.variational.CholeskyVariationalDistribution(
            num_inducing_points=grid_size, batch_shape=torch.Size([num_dim])
        )
        variational_strategy = gpytorch.variational.MultitaskVariationalStrategy(
            gpytorch.variational.GridInterpolationVariationalStrategy(
                self, grid_size=grid_size, grid_bounds=[grid_bounds],
                variational_distribution=variational_distribution,
            ), num_tasks=num_dim,
        )
        super().__init__(variational_strategy)

        self.covar_module = gpytorch.kernels.ScaleKernel(
            gpytorch.kernels.RBFKernel(
                lengthscale_prior=gpytorch.priors.SmoothedBoxPrior(
                    math.exp(-1), math.exp(1), sigma=0.1, transform=torch.exp
                )
            )
        )
        self.mean_module = gpytorch.means.ConstantMean()
        self.grid_bounds = grid_bounds

    def forward(self, x):
        mean = self.mean_module(x)
        covar = self.covar_module(x)
        return gpytorch.distributions.MultivariateNormal(mean, covar)

class DKLModel(gpytorch.Module):
    def __init__(self, feature_extractor, num_dim, likelihood, grid_bounds=(-10., 10.)):
        super(DKLModel, self).__init__()
        self.feature_extractor = feature_extractor
        self.gp_layer = GaussianProcessLayer(num_dim=num_dim, grid_bounds=grid_bounds)
        self.grid_bounds = grid_bounds
        self.num_dim = num_dim
        self.likelihood = likelihood
        self.drop = nn.Dropout(0.5)

    def forward(self, x):
        features = self.feature_extractor(x)
        features = gpytorch.utils.grid.scale_to_bounds(features, self.grid_bounds[0], self.
            grid_bounds[1])
        features = features.transpose(-1, -2).unsqueeze(-1)
        res_gp = self.gp_layer(features)
        return res_gp

def train(args, model, likelihood, mll, device, train_loader, optimizer, epoch):
    model.train()

    total_loss = 0
    loss_fn = nn.NLLLoss()

    minibatch_iter = tqdm.notebook.tqdm(train_loader, desc=f"(Epoch {epoch}) Minibatch")
    with gpytorch.settings.num_likelihood_samples(8):
      for data, target in minibatch_iter:
          data = data.to(device).float()
          target = target.to(device)
          optimizer.zero_grad()
          output = model(data)
          loss = F.nll_loss(output, target)
          total_loss += loss.item()
          loss.backward()
          optimizer.step()
          minibatch_iter.set_postfix(loss=loss.item())
    total_loss /= len(train_loader.dataset)
    return total_loss

def test(model, likelihood, mll, device, test_loader, epoch):
    model.eval()
    #likelihood.eval()
    test_loss = 0
    correct = 0
    minibatch_iter = tqdm.notebook.tqdm(test_loader, desc=f"(Epoch {epoch}) Minibatch")
    with torch.no_grad(), gpytorch.settings.num_likelihood_samples(16):
        for data, target in minibatch_iter:
            data = data.to(device).float()
            target = target.to(device)
            output = model(data)
            loss = F.nll_loss(output, target)
            test_loss += loss.item()
            pred = output.argmax(dim=1, keepdim=True)
            correct += pred.eq(target.view_as(pred)).sum().item()

    test_loss /= len(test_loader.dataset)

    return test_loss, correct / len(test_loader.dataset)

num_dim = 10
num_classes = train_loader.dataset.num_classes
model = CNN(num_classes)

model = model.to(device)
optimizer = SGD(model.parameters(), lr=args['learning_rate'], momentum=0.9, nesterov=True,
    weight_decay=0)
scheduler = StepLR(optimizer, step_size=args['step_size'], gamma=args['gamma'])
print(num_classes)

for epoch in range(1, args['n_epochs'] + 1):
  train_loss = train(args, model, None, None, device, train_loader, optimizer, epoch)
  test_loss, acc = test(model, None, None, device, test_loader, epoch)
  print('\n==> Epoch: {}, Train Loss: {:.4e}, Test Loss: {:.4e}, Test Acc: {:.4e}'.format(
      epoch, train_loss, test_loss, acc))
  scheduler.step()

"""Individual Testing"""

model.eval()
examples = iter(test_loader)
example_data, example_targets = next(examples)
data, target = example_data.to(device).float().view(-1,3,256,256), example_targets.to(device
    )
output = model(data)
observed_pred = likelihood(output)
preds = observed_pred.probs.mean(0).cpu()
pred = preds.argmax(-1)
pred_distribution = preds.cpu().detach().numpy()
var = observed_pred.probs.view((-1,train_dataset.num_classes)).var(1).cpu().detach().numpy()
print(pred.eq(example_targets.view_as(pred)).cpu().sum().item() / float(len(pred)))

idx = 0
fig = plt.figure()
plt.tight_layout()
im = example_data[idx].view((256,256,-1)) * 0.5 + 0.5
plt.imshow(im)
plt.title("Ground Truth: {}".format(example_targets[idx]))
plt.xticks([])
```

```python
plt.yticks([])
plt.show()

plt.figure()
plt.bar(np.arange(train_dataset.num_classes), pred_distribution[idx].squeeze(), yerr=var[idx
    ].squeeze())
plt.show()

pred[idx].item(), example_targets[idx].item()

train_dataset.categories[pred[idx].item()], train_dataset.categories[example_targets[idx].
    item()]

pred_distribution[idx]

var
```

11 Appendix: DKL CNN-GP Code

```python
! pip install gpytorch
import torch
import torchvision
from torchvision import datasets, transforms
import torchvision.models as models
import matplotlib.pyplot as plt
import torch.nn as nn
import torch.nn.functional as F
from torch.optim import SGD, Adam
import torch.optim as optim
from torch.optim.lr_scheduler import StepLR
import tqdm
from torch.optim.lr_scheduler import MultiStepLR
import gpytorch
import math
import numpy as np
import os
from PIL import Image

# Commented out IPython magic to ensure Python compatibility.
from google.colab import drive
drive.mount('/content/drive/')
# % cd /content/drive/My Drive/Colab Notebooks/UCMerced_LandUse
use_cuda = torch.cuda.is_available()
device = torch.device("cuda" if use_cuda else "cpu")
kwargs = {'num_workers': 1, 'pin_memory': True} if use_cuda else {}
args = {'log_interval': 1, 'learning_rate': 1e-3, 'step_size': 75, 'gamma': 0.1, 'n_epochs':
     50, 'batch_size_train': 128, 'batch_size_test': 64}
torch.manual_seed(1)
print(use_cuda)

class Dataset(torch.utils.data.Dataset):
  def __init__(self, mode, split, noise=False):
    path = './Images/'
    self.noise = noise
    categories = os.listdir(path)
    imgs = []
    target = []
    for category_idx in (range(len(categories))):
      img_categories = os.listdir(path + categories[category_idx])
      for img_idx in (range(len(img_categories))):
        im_path = path + categories[category_idx] + '/' + img_categories[img_idx]
        imgs.append(im_path)
        target.append(category_idx)
    self.imgs = np.array(imgs)
    self.target = np.array(target)
    self.img_exceptions = [130, 183, 209, 243, 396, 504, 505, 506, 507, 622, 623, 624, 633,
                            770, 788, 858, 861, 863, 864, 865, 866, 867, 868, 869, 870, 915,
                            935, 945, 993, 1055, 1060, 1077, 1122, 1145, 1146, 1308, 1320,
                            1699, 1714, 1736, 1857, 2060, 2062, 2063]
    self.img_idxs = []
    for idx in range(len(imgs)):
      if idx not in self.img_exceptions:
        self.img_idxs.append(idx)

    self.img_idxs = np.array(self.img_idxs)
    np.random.shuffle(self.img_idxs)

    self.imgs = self.imgs[self.img_idxs]
    self.target = self.target[self.img_idxs]
    self.num_classes = max(self.target)
    self.categories = categories

    split_idx = int(len(self.imgs) * split)
    if mode == 'train':
        self.imgs = self.imgs[:split_idx]
        self.target = self.target[:split_idx]
    elif mode == 'test':
        self.imgs = self.imgs[split_idx:]
        self.target = self.target[split_idx:]

  def __len__(self):
    return len(self.imgs)

  def __getitem__(self, index):
    idx = index

    im_path = self.imgs[idx]
    im = Image.open(im_path)
    im = np.array(im)
    if self.noise:
        pois = 50
        im = np.random.poisson(im / 255. * pois) / pois * 255

    im = im / 255.

    means = [0.485, 0.456, 0.406]
    stds = [0.229, 0.224, 0.225]

    means = [0.5, 0.5, 0.5]
    stds = [0.5, 0.5, 0.5]

    for ch in range(3):
        im[:,:,ch] = (im[:,:,ch] - means[ch]) / stds[ch]

    return im.reshape((3,256,256)), self.target[idx]

train_dataset = Dataset('train', 0.7)
test_dataset = Dataset('test', 0.7)

train_loader = torch.utils.data.DataLoader(train_dataset, shuffle = True)
test_loader = torch.utils.data.DataLoader(test_dataset, shuffle = True)

test_dataset_noise = Dataset('test', 0.7, noise = True)
test_loader_noise = torch.utils.data.DataLoader(test_dataset_noise, shuffle = True)

class CNN(nn.Module):
    def __init__(self, num_features):
        super(CNN, self).__init__()
        self.conv1 = nn.Conv2d(3, 6, 5)
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(6, 16, 5)
        self.fc1 = nn.Linear(59536, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, num_features)

    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = x.view(-1, 59536)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x

class GaussianProcessLayer(gpytorch.models.ApproximateGP):
    def __init__(self, num_dim, grid_bounds=(-10., 10.), grid_size=64):
        variational_distribution = gpytorch.variational.CholeskyVariationalDistribution(
            num_inducing_points=grid_size, batch_shape=torch.Size([num_dim])
        )
        variational_strategy = gpytorch.variational.MultitaskVariationalStrategy(
            gpytorch.variational.GridInterpolationVariationalStrategy(
                self, grid_size=grid_size, grid_bounds=[grid_bounds],
                variational_distribution=variational_distribution,
            ), num_tasks=num_dim,
        )
        super().__init__(variational_strategy)

        self.covar_module = gpytorch.kernels.ScaleKernel(
            gpytorch.kernels.RBFKernel(
                lengthscale_prior=gpytorch.priors.SmoothedBoxPrior(
                    math.exp(-1), math.exp(1), sigma=0.1, transform=torch.exp
                )
            )
        )
        self.mean_module = gpytorch.means.ConstantMean()
        self.grid_bounds = grid_bounds

    def forward(self, x):
        mean = self.mean_module(x)
        covar = self.covar_module(x)
        return gpytorch.distributions.MultivariateNormal(mean, covar)

class DKLModel(gpytorch.Module):
    def __init__(self, feature_extractor, num_dim, likelihood, grid_bounds=(-10., 10.)):
        super(DKLModel, self).__init__()
        self.feature_extractor = feature_extractor
        self.gp_layer = GaussianProcessLayer(num_dim=num_dim, grid_bounds=grid_bounds)
        self.grid_bounds = grid_bounds
        self.num_dim = num_dim
        self.likelihood = likelihood
        self.drop = nn.Dropout(0.5)

    def forward(self, x):
        features = self.feature_extractor(x)
        features = gpytorch.utils.grid.scale_to_bounds(features, self.grid_bounds[0], self.
            grid_bounds[1])
        features = features.transpose(-1, -2).unsqueeze(-1)
        res_gp = self.gp_layer(features)
        return res_gp

def train(args, model, likelihood, mll, device, train_loader, optimizer, epoch):
    model.train()
    likelihood.train()

    total_loss = 0
    loss_fn = nn.NLLLoss()

    minibatch_iter = tqdm.notebook.tqdm(train_loader, desc=f"(Epoch {epoch}) Minibatch")
    with gpytorch.settings.num_likelihood_samples(8):
      for data, target in minibatch_iter:
          data = data.to(device).float()
          target = target.to(device)
          optimizer.zero_grad()
          output = model(data)
          loss = -mll(output, target)
          total_loss += loss.item()
          loss.backward()
          optimizer.step()
          minibatch_iter.set_postfix(loss=loss.item())
    total_loss /= len(train_loader.dataset)
    return total_loss

def test(model, likelihood, mll, device, test_loader, epoch):
    model.eval()
    likelihood.eval()
    test_loss = 0
    correct = 0
    minibatch_iter = tqdm.notebook.tqdm(test_loader, desc=f"(Epoch {epoch}) Minibatch")
    with torch.no_grad(), gpytorch.settings.num_likelihood_samples(16):
        for data, target in minibatch_iter:
            data = data.to(device).float()
            target = target.to(device)
            output = model(data)
            loss = -mll(output, target)
            test_loss += loss.item()
            pred = likelihood(output).probs.mean(0).argmax(-1)
            correct += pred.eq(target.view_as(pred)).sum().item()

    test_loss /= len(test_loader.dataset)

    return test_loss, correct / len(test_loader.dataset)

num_dim = 100
num_classes = train_loader.dataset.num_classes
#resnet18 = models.resnet18(pretrained=True)
#NN = resnet18
NN = CNN(num_dim)
likelihood = gpytorch.likelihoods.SoftmaxLikelihood(num_features=num_dim, num_classes=num_
    classes)
model = DKLModel(NN, num_dim, likelihood)

model = model.to(device)
likelihood = likelihood.to(device)

learnable_params = [
    { 'params': model.feature_extractor.parameters(), 'weight_decay': 1e-6},
    {'params': model.gp_layer.hyperparameters(), 'lr': args['learning_rate'] * 0.01},
    {'params': model.gp_layer.variational_parameters()},
    {'params': model.likelihood.parameters()},
]

optimizer = SGD(learnable_params, lr=args['learning_rate'], momentum=0.9, nesterov=True,
    weight_decay=0)
scheduler = StepLR(optimizer, step_size=args['step_size'], gamma=args['gamma'])

mll = gpytorch.mlls.PredictiveLogLikelihood(likelihood, model.gp_layer, len(train_loader))
print(num_classes)

for epoch in range(1, args['n_epochs'] + 1):
  train_loss = train(args, model, likelihood, mll, device, train_loader, optimizer, epoch)
  test_loss, acc = test(model, likelihood, mll, device, test_loader, epoch)
  test_loss_noise, acc_noise = test(model, likelihood, mll, device, test_loader_noise, epoch
      )
  print('\n==> Epoch: {}, Train Loss: {:.4e}, Test Loss: {:.4e}, Test Acc: {:.4e}, Test Loss
       Noise: {:.4e}, Test Acc Noise: {:.4e}'.format(epoch, train_loss, test_loss, acc, test
      _loss_noise, acc_noise))
  scheduler.step()
```

```python

"""Individual Testing"""

model.eval()
examples = iter(test_loader_noise)
example_data, example_targets = next(examples)
data, target = example_data.to(device).float().view(-1,3,256,256), example_targets.to(device
    )
output = model(data)
observed_pred = likelihood(output)
preds = observed_pred.probs.mean(0).cpu()
pred = preds.argmax(-1)
pred_distribution = preds.cpu().detach().numpy()
var = observed_pred.probs.view((-1,train_dataset.num_classes)).var(1).cpu().detach().numpy()
print(pred.eq(example_targets.view_as(pred)).cpu().sum().item() / float(len(pred)))

idx = 0
fig = plt.figure()
plt.tight_layout()
im = example_data[idx].view((256,256,-1)) * 0.5 + 0.5
plt.imshow(im)
plt.title("Ground Truth: {}".format(example_targets[idx]))
plt.xticks([])
plt.yticks([])
plt.show()

plt.figure()
plt.bar(np.arange(train_dataset.num_classes), pred_distribution[idx].squeeze(), yerr=var[idx
    ].squeeze())
plt.show()

pred[idx].item(), example_targets[idx].item()

train_dataset.categories[pred[idx].item()], train_dataset.categories[example_targets[idx].
    item()]

pred_distribution[idx]

var
```