ECE-471 Selected Topics in Machine Learning
Prof. Curro
Midterm Project

Evan Bubniak, Zhang Jinhan

October 28, 2019

1 Summary

The purpose of this project is to reproduce a subset of experimental results from the paper *Understanding deep learning requires rethinking generalization* (Zhang et al. 2016), specifically Figure 1, which plots the average loss against the number of training steps for various types of example/label corruption. The experiment demonstrates that various kinds of label corruption do not prevent complete memorization of the training dataset, as long as the model is sufficiently large, and that the convergence of the model to 100% accuracy is shifted by only a constant factor when data corruption is introduced.

The paper does not specify certain hyperparameters, specifically number of epochs and batch size. This proved problematic as the interplay between these two variables affects the number of steps needed to converge. Furthermore, although five models are specified as part of the CIFAR-10 experimental set-up, it is not specified if only one model was used to generate Figure 1, or if multiple model results were averaged together.

We conducted several tests to reconstruct and verify the experimental setup. Specifically, we rebuilt each of the five models described in Table 1 of the original paper: two minified InceptionV3 models respectively with and without batch normalization, a minified AlexNet, and two multi-layer perceptions with 512 units and, respectively, one and three hideen layers. These models were defined in Keras and are stated in the appendix.

The experimental setup involved, in total, the training and analysis of 25 models,

corresponding to five models each trained on five datasets. Training was performed on the KAHAN server.

Our results show strong agreement with Figure 1 for the first model specified, the minified InceptionV3, and strong divergence in all other models. The two models are displayed side-by-side.

2 Results

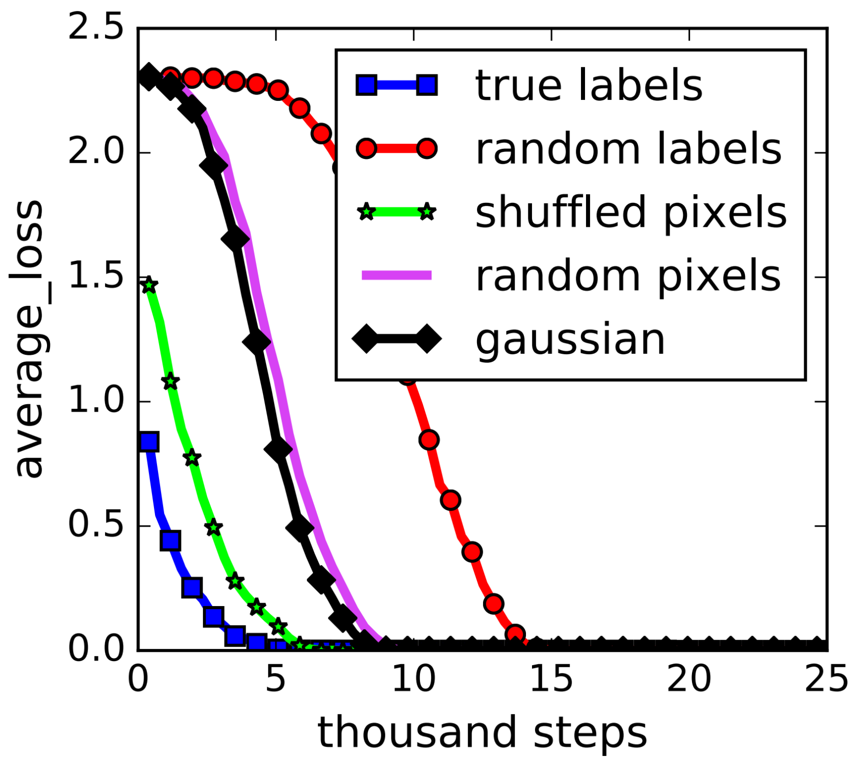

Figure 1: Juxtaposition of the noisy data points, the noiseless sinewave they are based on, and the manifold of the stochastic gradient descent regression model.

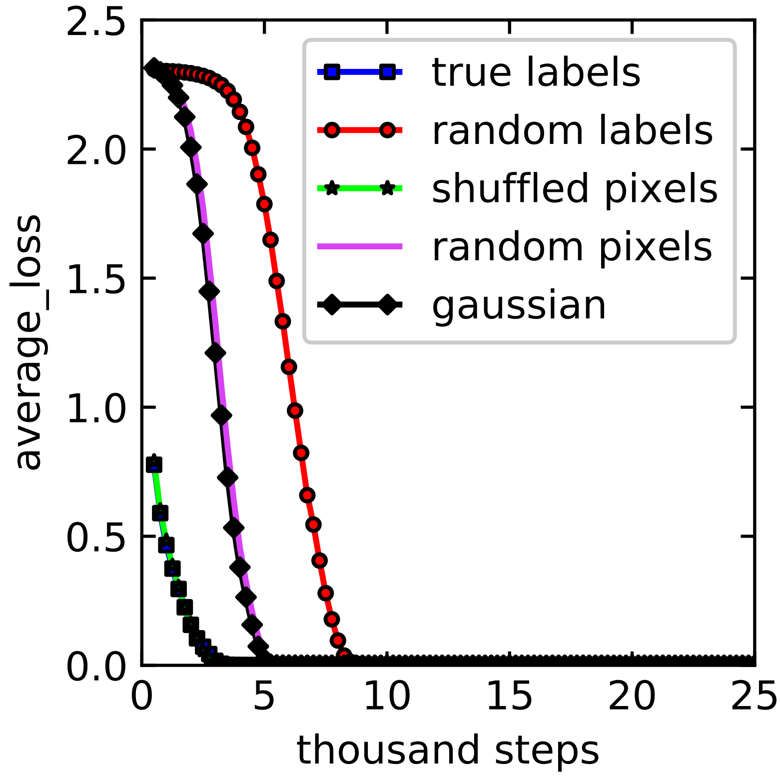

Figure 2: A plot of each of the basis functions, with the weights and intercept removed.

3 Appendix

3.1 Main Loop

```python
import tensorflow.keras as keras
from tensorflow.keras.datasets import cifar10
from utils import *
from math import ceil
import sys

BATCH_SIZE = 125
NUM_SAMPLES = 50000
NUM_EPOCHS = 100
STEPS_PER_EPOCH = ceil(NUM_SAMPLES / BATCH_SIZE)

def get_model(model_code):
    if model_code == 1:
        return MiniInceptionV3(
            input_shape = X.shape[1:],
            num_labels = 10
            )
    elif model_code == 2:
        return MiniInceptionV3(
            input_shape = X.shape[1:],
            num_labels = 10,
            use_batch_norm = False
            )
    elif model_code == 3:
        return AlexNet(
            input_shape = X.shape[1:],
            num_labels = 10
        )
    elif model_code == 4:
        return MLP(
            input_shape = X.shape[1:],
            num_labels = 10,
            num_hidden_layers = 1
        )
```

```python
    elif model_code == 5:
        return MLP(
            input_shape = X.shape[1:],
            num_labels = 10,
            num_hidden_layers = 3
        )

model_codes = [1, 2, 3, 4, 5]

(X_train, y_train), (X_test, y_test) = cifar10.load_data()

X_train = preprocess_input(X_train)
X_test = preprocess_input(X_test)
y_train = preprocess_labels(y_train)
y_test = preprocess_labels(y_test)

CORRUPTION_TYPE = ["true_labels", "random_labels",
                   "shuffled_pixels", "random_pixels", "gaussian"]
true_inputs = [X_train, y_train, X_test, y_test]
random_labels = [X_train, randomize_labels(y_train.shape[0], 10),
                 X_test, randomize_labels(y_test.shape[0], 10)]
shuffled_pixels = [shuffle_pixels(X_train), y_train,
                   shuffle_pixels(X_test), y_test]
random_pixels = [randomize_pixels(X_train), y_train,
                 randomize_pixels(X_test), y_test]
gaussian = [create_gaussian_noise(X_train), y_train,
            create_gaussian_noise(X_test), y_test]
DATA_INPUTS = [true_inputs, random_labels, shuffled_pixels,
               random_pixels, gaussian]

for model_code in model_codes:
    for job_name, data_input in zip(CORRUPTION_TYPE, DATA_INPUTS):
        X = data_input[0]
        y = data_input[1]
        test_X = data_input[2]
        test_y = data_input[3]
        model = get_model(model_code)
        model.compile()
```

```
    model.fit(*data_input, NUM_EPOCHS, job_name, BATCH_SIZE)
    model.evaluate(test_X, test_y)
```

3.2 Data Preprocessing

```
from tensorflow.image import per_image_standardization
from tensorflow.keras.utils import to_categorical

def normalize(x_input):
    return x_input/255

def crop(x_input):
    return x_input[:, 2:-2, 2:-2, :]

def preprocess_input(x_input):
    x_out = normalize(x_input)
    x_out = crop(x_out)
    x_out = per_image_standardization(x_out)
    return x_out

def preprocess_labels(y_input):
    return to_categorical(y_input)
```

3.3 Data Corruption

```
import numpy as np
np.random.seed(31415)

def randomize_labels(num_labels, num_unique_labels):
    random_labels = np.random.randint(low=0,
                                      high=num_unique_labels,
                                      size=num_labels)
    categorical_random_labels = np.eye(num_unique_labels)[random_labels]
    return categorical_random_labels

def shuffle_image_pixels(image_data):
```

```
    shuffled_image_pixels = np.copy(image_data)
    np.random.shuffle(shuffled_image_pixels)
    np.random.seed(31415)
    return shuffled_image_pixels

def shuffle_pixels(normalized_pixel_data):
    shuffled_pixel_data = np.copy(normalized_pixel_data)
    np.apply_along_axis(shuffle_image_pixels, 1,
        shuffled_pixel_data)
    return shuffled_pixel_data

def randomize_pixels(normalized_pixel_data):
    randomized_pixel_data = np.random.uniform(low = 0,
        high = 1.0,
        size=(normalized_pixel_data.shape))
    return randomized_pixel_data

def create_gaussian_noise(normalized_pixel_data):
    pixel_data = normalized_pixel_data.numpy().astype('float32')
    mean = pixel_data.mean(axis=0)
    std = pixel_data.std(axis=0)
    noisy_pixel_data = np.random.normal(mean, std,
        normalized_pixel_data.shape)
    np.clip(noisy_pixel_data, a_min=0, a_max = 1,
        out=noisy_pixel_data)
    return noisy_pixel_data
```

3.4 Model Definitions

```
import tensorflow.keras as keras
import os

RATE_DECAY_FACTOR_PER_EPOCH = 0.95
MOMENTUM_PARAMETER = 0.9

everything_in_dir = os.listdir(os.getcwd())
folders_in_dir = filter(
```

```
    lambda f: os.path.isdir(f) and "output" in f,
    everything_in_dir)
max_folder_num = 0
for folder in folders_in_dir:
    folder_num = folder[(folder.find("_") + 1):]
    max_folder_num = max(int(folder_num), max_folder_num)
OUTPUT_DIR = "output_{}".format(max_folder_num + 1)
if not os.path.exists(OUTPUT_DIR):
    os.mkdir(OUTPUT_DIR)

if keras.backend.image_data_format() == 'channels_first':
    CHANNEL_AXIS = 1
else:
    CHANNEL_AXIS = 3

if keras.backend.image_data_format() == 'channels_first':
    BATCHNORM_AXIS = 1
else:
    BATCHNORM_AXIS = 3

class MidtermModel:
    def __init__(self, weight_decay = None):
        self.model_name = "model"
        self.model = keras.models.Model()
        self.initial_learning_rate = 0.1
        self.weight_decay = weight_decay

    def compile(self):

        sgd = keras.optimizers.SGD(
            learning_rate = self.initial_learning_rate,
            momentum = MOMENTUM_PARAMETER,
            nesterov = False
            )

        self.model.compile(
            loss="categorical_crossentropy",
            optimizer=sgd,
```

```
        metrics=["acc", "top_k_categorical_accuracy"])

    print("Finished compiling")
    self.model.summary()

def fit(self, X_train, y_train, X_val, y_val,
    num_epoch, data_name, batch_size):
    run_name = "{}-{}".format(self.model_name, data_name)
    weights_file_name = "{}-weights.h5".format(run_name)
    weights_output_path = os.path.join(OUTPUT_DIR, weights_file_name)
    log_file_name = "{}.csv".format(run_name)
    log_output_path = os.path.join(OUTPUT_DIR, log_file_name)

    callbacks = [
            keras.callbacks.LearningRateScheduler(
                self.learning_rate_schedule,
                verbose = 0),
            keras.callbacks.ModelCheckpoint(weights_output_path,
                monitor="acc",
                save_best_only=True,
                verbose=1),
            keras.callbacks.CSVLogger(
                log_output_path)
            ]

    self.model_log = self.model.fit(X_train, y_train,
        batch_size = batch_size,
        epochs = num_epoch,
        verbose = 1,
        validation_data = (X_val, y_val),
        callbacks = callbacks)

    return self.model_log

def evaluate(self, X_test, y_test):
    self.model.evaluate(X_test, y_test, verbose = 1)

def learning_rate_schedule(self, epoch_num):
```

```python
        return self.initial_learning_rate * \
            (RATE_DECAY_FACTOR_PER_EPOCH)**(epoch_num)

class MiniInceptionV3(MidtermModel):
    def __init__(self, input_shape, num_labels=10, use_batch_norm = True):
        super(MiniInceptionV3, self).__init__()
        self.initial_learning_rate = 0.1
        self.model_name = "MiniInceptionV3"
        if not use_batch_norm:
            self.model_name += "_without_BatchNorm"
        self.use_batch_norm = use_batch_norm
        input_layer = keras.layers.Input(shape = input_shape)
        x = self.conv_module(input_layer, 96,
            kernel_size = (3,3),
            strides = (1,1))
        x = self.inception_module(x, 32, 32)
        x = self.inception_module(x, 32, 48)
        x = self.downsample_module(x, 80)
        x = self.inception_module(x, 112, 48)
        x = self.inception_module(x, 96, 64)
        x = self.inception_module(x, 80, 80)
        x = self.inception_module(x, 48, 96)
        x = self.downsample_module(x, 96)
        x = self.inception_module(x, 176, 160)
        x = self.inception_module(x, 176, 160)
        x = keras.layers.GlobalAveragePooling2D(
            data_format = keras.backend.image_data_format())(x)
        x = keras.layers.Dense(
            num_labels, activation='softmax', name='predictions')(x)

        self.model = keras.models.Model(input_layer, x, name=self.model_name)

    def conv_module(self, input_layer, filters, kernel_size,
        padding='same', strides=(1, 1)):
        x = keras.layers.Conv2D(
            filters,
            kernel_size = kernel_size,
            strides=strides,
```

```
            padding=padding,
            use_bias=False)(input_layer)
        if self.use_batch_norm:
            x = keras.layers.BatchNormalization(
                axis=BATCHNORM_AXIS, scale=False)(x)
        x = keras.layers.Activation('relu')(x)
        return x

    def inception_module(self, input_layer, filters_1, filters_3):
        conv_module1 = self.conv_module(
            input_layer, filters = filters_1,
            kernel_size = (1, 1), strides = (1, 1))
        conv_module3 = self.conv_module(
            input_layer, filters = filters_3,
            kernel_size = (3, 3), strides = (1, 1))

        return keras.layers.concatenate(
            [conv_module1,
            conv_module3],
            axis = CHANNEL_AXIS)

    def downsample_module(self, input_layer, filters):
        max_pooling = keras.layers.MaxPooling2D(
            pool_size = (3,3), strides = (2,2),
            padding = 'same')(input_layer)
        conv_module3 = self.conv_module(
            input_layer, filters,
            kernel_size = (3, 3), strides = (2,2))

        return keras.layers.concatenate(
            [conv_module3,
            max_pooling],
            axis = CHANNEL_AXIS)

class LocalResponseNormalization(keras.layers.Layer):
    '''
    Code sample adapted from "Deep Learning with Keras" by Gulli and Pal
    '''
```

```python
    def __init__(self, n=5, alpha=0.0005, beta=0.75, k=2, **kwargs):
        self.n = n
        self.alpha = alpha
        self.beta = beta
        self.k = k
        super(LocalResponseNormalization, self).__init__(**kwargs)

    def build(self, input_shape):
        self.shape = input_shape
        super(LocalResponseNormalization, self).build(input_shape)

    def call(self, x, mask=None):
        if keras.backend.image_data_format() == 'channels_first':
            _, f, r, c = self.shape
        else:
            _, r, c, f = self.shape
        squared = keras.backend.square(x)
        pooled = keras.backend.pool2d(squared,
            (self.n,self.n), strides = (1,1),
            padding = "same", pool_mode = "avg")
        summed = keras.backend.sum(pooled, axis=CHANNEL_AXIS, keepdims = True)
        averaged = self.alpha * keras.backend.repeat_elements(
            summed, f, axis=CHANNEL_AXIS)
        denom = keras.backend.pow(self.k + averaged, self.beta)
        return x / denom

    def get_output_shape_for(self, input_shape):
        return input_shape


class AlexNet(MidtermModel):
    def __init__(self, input_shape, num_labels=10):
        super(AlexNet, self).__init__()
        self.initial_learning_rate = 0.01
        self.model_name = "AlexNet"
        input_layer = keras.layers.Input(shape = input_shape)
        x = self.small_module(input_layer, filters = 96)
        x = self.small_module(x, filters = 256)
```

```
        x = keras.layers.Flatten()(x)
        x = keras.layers.Dense(384, activation = 'relu')(x)
        x = keras.layers.Dense(192, activation = 'relu')(x)
        x = keras.layers.Dense(
            num_labels,
            activation='softmax',
            name='predictions')(x)

        self.model = keras.models.Model(input_layer, x, name=self.model_name)

    def small_module(self, input_layer, filters = 96):
        x = keras.layers.Conv2D(
            filters,
            kernel_size = (5, 5),
            padding = 'valid')(input_layer)
        x = keras.layers.MaxPooling2D(
            pool_size = (3, 3), padding = "valid")(x)
        x = LocalResponseNormalization()(x)
        return x

class MLP(MidtermModel):
    def __init__(self, input_shape, num_labels=10,
                 num_hidden_layers = 1,
                 num_hidden_units = 512):
        super(MLP, self).__init__()
        self.model_name = "MLP_{}x{}".format(
            num_hidden_layers, num_hidden_units)
        input_layer = keras.layers.Input(shape = input_shape)
        x = keras.layers.Flatten()(input_layer)
        for hidden_layer in range(num_hidden_layers):
            x = keras.layers.Dense(num_hidden_units)(x)
            x = keras.layers.Activation("relu")(x)
        x = keras.layers.Dense(
            num_labels,
            activation = "softmax",
            name = "predictions")(x)
        self.model = keras.models.Model(
            input_layer, x, name=self.model_name)
```

3.5 Matplotlib configuation

```python
"""
Plot of learning curves
true labels: blue, with blue square dots
random labels - red, with red circular dots
shuffled pixels: green, with green star 5-point star dots
random pixels: purple, with no dots
gaussian: black, with black diamond dots
"""
import matplotlib.pyplot as plt
import pandas as pd
import os
import argparse
from math import ceil

label_markers = ['true', 'random_labels', 'shuffled',
                 'random_pixels', 'gaussian']
model_names = ["MiniInceptionV3",
               "MiniInceptionV3_without_BatchNorm",
               "AlexNet", "MLP_1x512", "MLP_3x512"]
model_correspondence = {
    "MiniInceptionV3": 1,
    "MiniInceptionV3_without_BatchNorm":2,
    "AlexNet":3,
    "MLP_1x512":4,
    "MLP_3x512":5
}
parser = argparse.ArgumentParser()
parser.add_argument("--model", nargs="*", default = model_names)
parser.add_argument("-e", "--num_epochs",
    nargs="?", type=int, default=100)
parser.add_argument("-b", "--batch_size",
    nargs="?", type=int, default=100)

parser.add_argument("-i", "--iterate", action = "store_true")
args = parser.parse_args()
```

```python
def plot_results(steps_per_epoch, models = model_names,
                 plot_name="output"):
    def filter_dir_files(file_name):
        conditionals = [".csv" in file_name]
        if ("MiniInceptionV3" in models and
            "MiniInceptionV3_without_BatchNorm" not in models and
            "BatchNorm" in file_name):
            return False
        else:
            conditionals.append(any([model_name in file_name
                for model_name in models]))
        return all(conditionals)

    all_data = []
    everything_in_dir = os.listdir(os.getcwd())
    folders_in_dir = filter(lambda f: os.path.isdir(f) and
                            "output" in f, everything_in_dir)
    max_folder_num = 1
    for folder in folders_in_dir:
        folder_num = folder[(folder.find("_") + 1):]
        max_folder_num = max(int(folder_num), max_folder_num)
    OUTPUT_DIR = "output_{}".format(max_folder_num)

    all_files_in_dir = list(filter(filter_dir_files,
                                   os.listdir(OUTPUT_DIR)))

    for label in label_markers:
        data_by_model = []
        for file_name in all_files_in_dir:
            if label in file_name:
                path = os.path.join(OUTPUT_DIR, file_name)
                data = pd.read_csv(path)
                data = data[:args.num_epochs]
                data_by_model.append(data)
        all_data.append(data_by_model)

    all_losses = []
    for data_corruption in all_data:
```

```
    avg_loss = pd.DataFrame({'average loss':
                             [0] * len(data_corruption[0])})
    avg_loss['epoch'] = avg_loss.index
    for model_results in data_corruption:
        avg_loss['average loss'] = avg_loss['average loss'] + \
            model_results["loss"]
    avg_loss['average loss'] = avg_loss['average loss'] / \
        len(data_corruption)
    all_losses.append(avg_loss)
for dataset in all_losses:
    dataset["thousand steps"] = \
        ((dataset["epoch"] + 1)*steps_per_epoch)/1000
true_label_format = [{'c': "blue", 'marker': 's',
                      'edgecolors': 'black'},
                     {'c': "blue"}]

random_label_format = [{'c': "red", 'marker': 'o',
                        'edgecolors': 'black'},
                       {'c': "red"}]

shuffled_pixel_format = [{'c': '#00ff00', 'marker': '*',
                          'edgecolors': 'black'},
                         {'c': '#00ff00'}]

random_pixel_format = [{'c': None, 'marker': None,
                        'edgecolors': None},
                       {'c': "#D742F4"}]

gaussian_format = [{'c': 'black', 'marker': 'D',
                    'edgecolors': 'black'},
                   {'c': "black"}]

formats = [true_label_format, random_label_format,
           shuffled_pixel_format, random_pixel_format,
           gaussian_format]
legend_names = ['true labels', 'random labels',
                'shuffled pixels', 'random pixels', 'gaussian']
```

```python
fig1 = plt.figure(figsize=(3,3))
i = 0
array_of_linmarks=[]
for dataset, data_format in zip(all_losses, formats):
    lin = None
    mark = None
    ax = plt.gca()
    z = 5 if i == 3 else 0
    if i != 3:
        mark = ax.scatter(dataset["thousand steps"].values[1:],
                    dataset["average loss"].values[1:],
                    s = 10,
                    zorder = 10,
                    **data_format[0])
    lin, = ax.plot(dataset["thousand steps"].values[1:],
            dataset["average loss"].values[1:],
            zorder = z,
            **data_format[1]
        )
    if mark:
        array_of_linmarks.append((lin, mark))
    else:
        array_of_linmarks.append((lin))
    i+=1
ax.tick_params(axis = 'both', direction = 'in',
               top = True, right = True)
leg = ax.legend(array_of_linmarks, legend_names,
                scatterpoints=2, framealpha = 1,
                scatteryoffsets=[0.5])
leg.set_zorder(20)
leg.get_frame().set_facecolor('w')
ax.set_xticks([0, 5, 10, 15, 20, 25])
ax.set_yticks([0, 0.5, 1, 1.5, 2.0, 2.5])

plt.draw()

plt.xlabel("thousand steps")
plt.ylabel("average_loss")
```

```python
    plt.xlim(0, 25)
    plt.ylim(0, 2.5)
    plt.tight_layout()
    fig1.savefig(os.path.join(OUTPUT_DIR, "{}.eps".format(plot_name)))
    fig1.savefig(os.path.join(OUTPUT_DIR, "{}.png".format(plot_name)))

if __name__ == "__main__":
    if args.iterate:
        for model in args.model:
            plot_results(
                ceil(50000/args.batch_size),
                models = [model],
                plot_name = model)
    else:
        plot_results(
            ceil(50000/args.batch_size),
            models = args.model)
```