

In [7]:

```
import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt
```

```
np.random.seed(4321)
```

In [8]:

```
(x_train, y_train), (x_test, y_test) = tf.keras.datasets.fashion_mnist.load_data()
x_val = x_train[50000:60000]
x_train = x_train[0:50000]
y_val = y_train[50000:60000]
y_train = y_train[0:50000]
x_train = x_train.astype(np.float32).reshape(-1,28,28,1) / 255.0
x_val = x_val.astype(np.float32).reshape(-1,28,28,1) / 255.0
x_test = x_test.astype(np.float32).reshape(-1,28,28,1) / 255.0
y_train = tf.one_hot(y_train, depth=10)
y_val = tf.one_hot(y_val, depth=10)
y_test = tf.one_hot(y_test, depth=10)
print(x_train.shape)
print(x_test.shape)
print(x_val.shape)
train_dataset = tf.data.Dataset.from_tensor_slices((x_train, y_train))
train_dataset = train_dataset.shuffle(buffer_size=1024).batch(128)
train_dataset_full = train_dataset.shuffle(buffer_size=1024).batch(len(train_data
et))
val_dataset = tf.data.Dataset.from_tensor_slices((x_val, y_val))
val_dataset = val_dataset.batch(128)
test_dataset = tf.data.Dataset.from_tensor_slices((x_test, y_test))
test_dataset = test_dataset.batch(128)
print(len(train_dataset))
print(len(test_dataset))
```

```
(50000, 28, 28, 1)
```

```
(10000, 28, 28, 1)
```

```
(10000, 28, 28, 1)
```

```
391
```

```
79
```

In [9]:

```

class BatchNormalization(tf.keras.layers.Layer):
    def __init__(self, batch_size, training=False):
        super(BatchNormalization, self).__init__()

        self.convexCoeff = 0.9
        self.numCalls = 0
        self.batch_size = batch_size
        self.training = training

        self.gamma = self.add_weight(name='gamma', shape=[self.batch_size,], ini-
alizer=tf.initializers.ones, trainable=True)
        self.beta = self.add_weight(name='beta', shape=[self.batch_size,], ini-
alizer=tf.initializers.zeros, trainable=True)
        self.mean = self.add_weight(name='mean', shape=[self.batch_size,], ini-
alizer=tf.initializers.zeros, trainable=False)
        self.var = self.add_weight(name='var', shape=[self.batch_size,], ini-
alizer=tf.initializers.zeros, trainable=False)

    def batch_norm(self, inputs, training):
        self.numCalls += 1

        axes = list(range(len(inputs.shape) - 1))
        mean = tf.reduce_mean(inputs, axes, keepdims=True)
        var = tf.reduce_mean(tf.math.squared_difference(inputs, tf.stop_gradient(
mean)), axes, keepdims=True)

        if training:
            norm = tf.add(tf.multiply(self.gamma, tf.divide(tf.subtract(inputs, me-
an), tf.sqrt(var+1e-7))), self.beta)

            mean = tf.squeeze(mean, axes)
            var = tf.squeeze(var, axes)

            moving_avg_mean = ((self.convexCoeff/self.numCalls)*mean) + (1-(self.c-
onvexCoeff/self.numCalls)*self.mean)
            moving_avg_var = ((self.convexCoeff/self.numCalls)*var) + (1-(self.c-
onvexCoeff/self.numCalls)*self.var)

            self.mean.assign(moving_avg_mean)
            self.var.assign(moving_avg_var)

        else:
            norm = tf.add(tf.multiply(self.gamma, tf.divide(tf.subtract(inputs, me-
an), tf.sqrt(var+1e-7))), self.beta)

        return norm

```

In [10]:

```

class ImageRecognitionCNN(tf.keras.Model):

    def __init__(self, num_classes, device='cpu:0', checkpoint_directory=None):
        """ Define the parameterized layers used during forward-pass, the device
            where you would like to run the computation (GPU, TPU, CPU) on and the
            checkpoint directory.

            Args:
                num_classes: the number of labels in the network.
                device: string, 'cpu:n' or 'gpu:n' (n can vary). Default, 'cpu:0'.
                checkpoint_directory: the directory where you would like to save o

            ...
            restore a model.
        """
        super(ImageRecognitionCNN, self).__init__()

        # Initialize layers
        self.conv1 = tf.keras.layers.Conv2D(64, 3, padding='same', activation=None)
        self.conv2 = tf.keras.layers.Conv2D(64, 3, padding='same', activation=None)
        self.pool1 = tf.keras.layers.MaxPool2D()
        self.conv3 = tf.keras.layers.Conv2D(64, 3, padding='same', activation=None)
        self.conv4 = tf.keras.layers.Conv2D(64, 3, padding='same', activation=None)
        # self.pool2 = tf.keras.layers.MaxPool2D()
        # self.conv5 = tf.keras.layers.Conv2D(64, 3, padding='same', activation=None)
        # self.pool2 = tf.keras.layers.MaxPool2D()
        # self.conv6 = tf.keras.layers.Conv2D(64, 3, 2, padding='same', activation=None)
        # self.conv7 = tf.keras.layers.Conv2D(64, 1, padding='same', activation=None)
        self.conv8 = tf.keras.layers.Conv2D(num_classes, 1, padding='same', activation=None)
        self.BN = BatchNormalization(64)

        # Define the device
        self.device = device

        # Define the checkpoint directory
        self.checkpoint_directory = checkpoint_directory
        self.acc = tf.keras.metrics.Accuracy()

    def predict(self, images, training):
        """ Predicts the probability of each class, based on the input sample.

            Args:
                images: 4D tensor. Either an image or a batch of images.
                training: Boolean. Either the network is predicting in
                    training mode or not.

```

```

"""
x = self.conv1(images)
x = self.BN.batch_norm(x, training)
x = tf.nn.relu(x)
x = self.pool1(x)
x = self.conv2(x)
x = self.BN.batch_norm(x, training)
x = tf.nn.relu(x)
x = self.pool1(x)
x = self.conv3(x)
x = self.BN.batch_norm(x, training)
x = tf.nn.relu(x)
x = self.pool1(x)
x = self.conv4(x)
x = self.BN.batch_norm(x, training)
x = tf.nn.relu(x)
x = self.pool1(x)
x = self.conv8(x)
#x = tf.nn.relu(x)
#print(x.shape)
x = tf.reshape(x, (-1, 1, 10))
#x = tf.keras.layers.Flatten(x)
return x

```

```

def loss_fn(self, images, target, training):
    """ Defines the loss function used during
        training.
    """
    preds = self.predict(images, training)
    #print(preds.shape)
    #print(target.shape)
    loss = tf.nn.softmax_cross_entropy_with_logits(labels=target, logits=preds
)
    return loss

```

```

def grads_fn(self, images, target, training):
    """ Dynamically computes the gradients of the loss value
        with respect to the parameters of the model, in each
        forward pass.
    """
    with tf.GradientTape() as tape:
        loss = self.loss_fn(images, target, training)
    return tape.gradient(loss, self.variables)

```

```

def restore_model(self):
    """ Function to restore trained model.
    """
    with tf.device(self.device):
        # Run the model once to initialize variables
        dummy_input = tf.constant(tf.zeros((1,48,48,1)))
        dummy_pred = self.predict(dummy_input, training=False)
        # Restore the variables of the model
        saver = tf.Saver(self.variables)

```

```

        saver.restore(tf.train.latest_checkpoint
                      (self.checkpoint_directory))

def save_model(self, global_step=0):
    """ Function to save trained model.
    """
    tf.Saver(self.variables).save(self.checkpoint_directory,
                                  global_step=global_step)

# def compute_accuracy(self, input_data):
#     """ Compute the accuracy on the input data.
#     """
#     with tf.device(self.device):
#         #acc = tf.metrics.Accuracy()
#         for step, (images, targets) in enumerate(input_data):
#             # Predict the probability of each class
#             #print(targets.shape)
#             logits = self.predict(images, training=False)
#             # Select the class with the highest probability
#             #print(logits.shape)
#             logits = tf.nn.softmax(logits)
#             logits = tf.reshape(logits, [-1, 10])
#             targets = tf.reshape(targets, [-1, 10])
#             preds = tf.argmax(logits, axis=1)

#             #m1.update_state
#             # Compute the accuracy
#             #print(preds.shape)
#             acc(tf.reshape(targets, preds))
#     return acc

def compute_accuracy_2(self, images, targets, training=False):
    """ Compute the accuracy on the input data.
    """
    with tf.device(self.device):

        # Predict the probability of each class
        logits = self.predict(images, training)
        # Select the class with the highest probability

        logits = tf.nn.softmax(logits)
        logits = tf.reshape(logits, [-1, 10])
        targets = tf.reshape(targets, [-1, 10])
        preds = tf.argmax(logits, axis=1)
        goal = tf.argmax(targets, axis=1)
        self.acc.update_state(goal, preds)
        # Compute the accuracy
        result = self.acc.result().numpy()
    return result

def fit_fc(self, training_data, eval_data, optimizer, num_epochs=500,
           early_stopping_rounds=10, verbose=10, train_from_scratch=False):
    """ Function to train the model, using the selected optimizer and
    for the desired number of epochs. You can either train from scratch
    or load the latest model trained. Early stopping is used in order to

```

mitigate the risk of overfitting the network.

Args:

*training_data: the data you would like to train the model on.
Must be in the tf.data.Dataset format.*

*eval_data: the data you would like to evaluate the model on.
Must be in the tf.data.Dataset format.*

optimizer: the optimizer used during training.

*num_epochs: the maximum number of iterations you would like to
train the model.*

*early_stopping_rounds: stop training if the loss on the eval
dataset does not decrease after n epochs.*

verbose: int. Specify how often to print the loss value of the net

work.

train_from_scratch: boolean. Whether to initialize variables of th

e
*the last trained model or initialize them
randomly.*

"""

if train_from_scratch==**False**:

 self.restore_model()

*# Initialize best loss. This variable will store the lowest loss on the
eval dataset.*

best_loss = 999

Initialize classes to update the mean loss of train and eval

train_loss = tf.keras.metrics.Mean('train_loss')

eval_loss = tf.keras.metrics.Mean('eval_loss')

acc_train = tf.keras.metrics.Mean('train_acc')

acc_val = tf.keras.metrics.Mean('val_acc')

Initialize dictionary to store the loss history

self.history = {}

self.history['train_loss'] = []

self.history['eval_loss'] = []

self.history['train_acc'] = []

self.history['val_acc'] = []

Begin training

with tf.device(self.device):

for i **in** range(num_epochs):

Training with gradient descent

#training_data_x = training_data.shuffle(buffer_size=1024).batch(1

28)

for step, (images, target) **in** enumerate(training_data):

 grads = self.grads_fn(images, target, **True**)

 optimizer.apply_gradients(zip(grads, self.variables))

Compute the loss on the training data after one epoch

for step, (images, target) **in** enumerate(training_data):

 loss = self.loss_fn(images, target, **False**)

 accuracy = self.compute_accuracy_2(images, target)

 acc_train(accuracy)

 train_loss(loss)

```

self.history['train_loss'].append(train_loss.result().numpy())
self.history['train_acc'].append(acc_train.result().numpy())
# Reset metrics
train_loss.reset_states()
acc_train.reset_states()

# Compute the loss on the eval data after one epoch
for step, (images, target) in enumerate(eval_data):
    loss = self.loss_fn(images, target, False)
    accuracy = self.compute_accuracy_2(images, target)
    acc_val(accuracy)
    eval_loss(loss)
self.history['eval_loss'].append(eval_loss.result().numpy())
self.history['val_acc'].append(acc_val.result().numpy())
# Reset metrics
eval_loss.reset_states()
acc_val.reset_states()

# Print train and eval losses
if (i==0) | ((i+1)%verbose==0):
    print('Train loss at epoch %d: ' % (i+1), self.history['train_loss'][-1])
    print('Train Acc at epoch %d: ' % (i+1), self.history['train_acc'][-1])
    print('Eval loss at epoch %d: ' % (i+1), self.history['eval_loss'][-1])
    print('Eval Acc at epoch %d: ' % (i+1), self.history['val_acc'][-1])

# Check for early stopping
if self.history['eval_loss'][-1]<best_loss:
    best_loss = self.history['eval_loss'][-1]
    count = early_stopping_rounds
else:
    count -= 1
if count==0:
    break

```

In [11]:

```

# Specify the path where you want to save/restore the trained variables.
checkpoint_directory = 'models_checkpoints/fmnist/'

# Use the GPU if available.
device = 'gpu:0'

# Define optimizer.
optimizer = tf.compat.v1.train.AdamOptimizer(learning_rate=1e-4)

# Instantiate model. This doesn't initialize the variables yet.
model = ImageRecognitionCNN(num_classes=10, device=device,
                             checkpoint_directory=checkpoint_directory)

#model = ImageRecognitionCNN(num_classes=7, device=device)

```

In [12]:

```
# Train model
model.fit_fc(train_dataset, val_dataset, optimizer, num_epochs=10,
             early_stopping_rounds=2, verbose=2, train_from_scratch=True)
```

```
Train loss at epoch 1: 0.41589844
Train Acc at epoch 1: 0.85882115
Eval loss at epoch 1: 0.4369065
Eval Acc at epoch 1: 0.85934436
Train loss at epoch 2: 0.32678548
Train Acc at epoch 2: 0.8661574
Eval loss at epoch 2: 0.35857725
Eval Acc at epoch 2: 0.872446
Train loss at epoch 4: 0.25572088
Train Acc at epoch 4: 0.88545567
Eval loss at epoch 4: 0.30858925
Eval Acc at epoch 4: 0.8888894
Train loss at epoch 6: 0.21292028
Train Acc at epoch 6: 0.8964649
Eval loss at epoch 6: 0.28468505
Eval Acc at epoch 6: 0.8988403
Train loss at epoch 8: 0.18097195
Train Acc at epoch 8: 0.90486234
Eval loss at epoch 8: 0.27084717
Eval Acc at epoch 8: 0.90677863
Train loss at epoch 10: 0.15379688
Train Acc at epoch 10: 0.91195726
Eval loss at epoch 10: 0.2628677
Eval Acc at epoch 10: 0.91359943
```

In []: