1. **Deep Learning.**
   1. **Build a DNN with five hidden layers of 100 neurons each, He initialization, and the ELU activation function.**

**To build a deep neural network (DNN) with 20 hidden layers of 100 neurons each, He initialization, and the ELU activation function, you can use the following code:**

**Python**

**from tensorflow import keras**

**model = keras.models.Sequential()**

**model.add(keras.layers.Flatten(input\_shape=[32, 32, 3]))**

**for \_ in range(20):**

**model.add(keras.layers.Dense(100,**

**activation="elu",**

**kernel\_initializer="he\_normal"))**

* 1. **Using Adam optimization and early stopping, try training it on MNIST but only on digits 0 to 4, as we will use transfer learning for digits 5 to 9 in the next exercise. You will need a softmax output layer with five neurons, and as always make sure to save checkpoints at regular intervals and save the final model so you can reuse it later.**

**To train the network on the CIFAR10 dataset using Nadam optimization and early stopping, you can use the following code:**

Python

**(X\_train\_full, y\_train\_full), (X\_test, y\_test) = keras.datasets.cifar10.load\_data()**

**X\_train = X\_train\_full[5000:]**

**y\_train = y\_train\_full[5000:]**

**X\_valid = X\_train\_full[:5000]**

**y\_valid = y\_train\_full[:5000]**

**model.add(keras.layers.Dense(10, activation="softmax"))**

**optimizer = keras.optimizers.Nadam(lr=5e-5)**

**model.compile(loss="sparse\_categorical\_crossentropy",**

**optimizer=optimizer,**

**metrics=["accuracy"])**

**early\_stopping\_cb = keras.callbacks.EarlyStopping(patience=20)**

**model.fit(X\_train, y\_train, epochs=100,**

**validation\_data=(X\_valid, y\_valid),**

**callbacks=[early\_stopping\_cb])**

* 1. **Tune the hyperparameters using cross-validation and see what precision you can achieve.**

**To add Batch Normalization and compare the learning curves, you can modify the model as follows:**

**Python**

**model = keras.models.Sequential()**

**model.add(keras.layers.Flatten(input\_shape=[32, 32, 3]))**

**model.add(keras.layers.BatchNormalization())**

**for \_ in range(20):**

**model.add(keras.layers.Dense(100, kernel\_initializer="he\_normal"))**

**model.add(keras.layers.BatchNormalization())**

**model.add(keras.layers.Activation("elu"))**

**model.add(keras.layers.Dense(10, activation="softmax"))**

* 1. **Now try adding Batch Normalization and compare the learning curves: is it converging faster than before? Does it produce a better model?**

**To replace Batch Normalization with SELU and make the necessary adjustments to ensure the network self-normalizes, you can modify the model as follows:**

**Python**

**model = keras.models.Sequential()**

**model.add(keras.layers.Flatten(input\_shape=[32, 32, 3]))**

**for \_ in range(20):**

**model.add(keras.layers.Dense(100,**

**activation="selu",**

**kernel\_initializer="lecun\_normal"))**

* 1. **Is the model overfitting the training set? Try adding dropout to every layer and try again. Does it help?**

**To regularize the model with alpha dropout and achieve better accuracy using MC Dropout, you can modify the model as follows:**

**Python**

**model = keras.models.Sequential()**

**model.add(keras.layers.Flatten(input\_shape=[32, 32, 3]))**

**for \_ in range(20):**

**model.add(keras.layers.Dense(100,**

**activation="selu",**

**kernel\_initializer="lecun\_normal"))**

**model.add(keras.layers.AlphaDropout(rate=0.1))**

**model.add(keras.layers.Dense(10, activation="softmax"))**

1. **Transfer learning.**
   1. **Create a new DNN that reuses all the pretrained hidden layers of the previous model, freezes them, and replaces the softmax output layer with a new one.**
   2. **Train this new DNN on digits 5 to 9, using only 100 images per digit, and time how long it takes. Despite this small number of examples, can you achieve high precision?**
   3. **Try caching the frozen layers, and train the model again: how much faster is it now?**
   4. **Try again reusing just four hidden layers instead of five. Can you achieve a higher precision?**
   5. **Now unfreeze the top two hidden layers and continue training: can you get the model to perform even better?**
2. **Pretraining on an auxiliary task.**

**In this exercise you will build a DNN that compares two MNIST digit images and predicts whether they represent the same digit or not. Then you will reuse the lower layers of this network to train an MNIST classifier using very little training data. Start by building two DNNs (let’s call them DNN A and B), both similar to the one you built earlier but without the output layer: each DNN should have five hidden layers of 100 neurons each, He initialization, and ELU activation. Next, add one more hidden layer with 10 units on top of both DNNs. To do this, you should use TensorFlow’s concat() function with axis=1 to concatenate the outputs of both DNNs for each instance, then feed the result to the hidden layer. Finally, add an output layer with a single neuron using the logistic activation function.**

**Let's download and load the dataset and display a few random samples from it:**

**mnist = tf.keras.datasets.mnist**

**(train\_images, train\_labels), (test\_images, test\_labels) = mnist.load\_data()**

**train\_images = (np.expand\_dims(train\_images, axis=-1)/255.).astype(np.float32)**

**train\_labels = (train\_labels).astype(np.int64)**

**test\_images = (np.expand\_dims(test\_images, axis=-1)/255.).astype(np.float32)**

**test\_labels = (test\_labels).astype(np.int64)**

**Our training set is made up of 28x28 grayscale images of handwritten digits.**

**Let's visualize what some of these images and their corresponding training labels look like.**

**plt.figure(figsize=(10,10))**

**random\_inds = np.random.choice(60000,36)**

**for i in range(36):**

**plt.subplot(6,6,i+1)**

**plt.xticks([])**

**plt.yticks([])**

**plt.grid(False)**

**image\_ind = random\_inds[i]**

**plt.imshow(np.squeeze(train\_images[image\_ind]), cmap=plt.cm.binary)**

**plt.xlabel(train\_labels[image\_ind])**

**comet\_model\_1.log\_figure(figure=plt)**

* 1. **Split the MNIST training set in two sets: split #1 should containing 55,000 images, and split #2 should contain contain 5,000 images. Create a function that generates a training batch where each instance is a pair of MNIST images picked from split #1. Half of the training instances should be pairs of images that belong to the same class, while the other half should be images from different classes. For each pair, the training label should be 0 if the images are from the same class, or 1 if they are from different classes.**

**Let's download and load the dataset and display a few random samples from it:**

**mnist = tf.keras.datasets.mnist**

**(train\_images, train\_labels), (test\_images, test\_labels) = mnist.load\_data()**

**train\_images = (np.expand\_dims(train\_images, axis=-1)/255.).astype(np.float32)**

**train\_labels = (train\_labels).astype(np.int64)**

**test\_images = (np.expand\_dims(test\_images, axis=-1)/255.).astype(np.float32)**

**test\_labels = (test\_labels).astype(np.int64)**

**Our training set is made up of 28x28 grayscale images of handwritten digits.**

**Let's visualize what some of these images and their corresponding training labels look like.**

**plt.figure(figsize=(10,10))**

**random\_inds = np.random.choice(60000,36)**

**for i in range(36):**

**plt.subplot(6,6,i+1)**

**plt.xticks([])**

**plt.yticks([])**

**plt.grid(False)**

**image\_ind = random\_inds[i]**

**plt.imshow(np.squeeze(train\_images[image\_ind]), cmap=plt.cm.binary)**

**plt.xlabel(train\_labels[image\_ind])**

**comet\_model\_1.log\_figure(figure=plt)**

* 1. **Train the DNN on this training set. For each image pair, you can simultaneously feed the first image to DNN A and the second image to DNN B. The whole network will gradually learn to tell whether two images belong to the same class or not.**

**Training the model 2.0**

**Earlier in the lab, we used the**[**fit**](https://www.tensorflow.org/api_docs/python/tf/keras/models/Sequential#fit)**function call to train the model. This function is quite high-level and intuitive, which is really useful for simpler models. As you may be able to tell, this function abstracts away many details in the training call, and we have less control over training model, which could be useful in other contexts.**

**As an alternative to this, we can use the [tf.GradientTape](https://www.tensorflow.org/api_docs/python/tf/GradientTape) class to record differentiation operations during training, and then call the [tf.GradientTape.gradient](https://www.tensorflow.org/api_docs/python/tf/GradientTape" \l "gradient) function to actually compute the gradients. You may recall seeing this in Lab 1 Part 1, but let's take another look at this here.**

**We'll use this framework to train our cnn\_model using stochastic gradient descent.**

***# Rebuild the CNN model***

**cnn\_model = build\_cnn\_model()**

**batch\_size = 12**

**loss\_history = mdl.util.LossHistory(smoothing\_factor=0.95) *# to record the evolution of the loss***

**plotter = mdl.util.PeriodicPlotter(sec=2, xlabel='Iterations', ylabel='Loss', scale='semilogy')**

**optimizer = tf.keras.optimizers.SGD(learning\_rate=1e-2) *# define our optimizer***

**comet\_ml.init(project\_name="6.s191lab2\_part1\_CNN2")**

**comet\_model\_3 = comet\_ml.Experiment()**

**if hasattr(tqdm, '\_instances'): tqdm.\_instances.clear() *# clear if it exists***

**for idx in tqdm(range(0, train\_images.shape[0], batch\_size)):**

***# First grab a batch of training data and convert the input images to tensors***

**(images, labels) = (train\_images[idx:idx+batch\_size], train\_labels[idx:idx+batch\_size])**

**images = tf.convert\_to\_tensor(images, dtype=tf.float32)**

***# GradientTape to record differentiation operations***

**with tf.GradientTape() as tape:**

***#'''TODO: feed the images into the model and obtain the predictions'''***

**logits = *# TODO***

***#'''TODO: compute the categorical cross entropy loss***

**loss\_value = tf.keras.backend.sparse\_categorical\_crossentropy('''TODO''', '''TODO''') *# TODO***

***# log the loss to comet***

**comet\_model\_3.log\_metric("loss", loss\_value.numpy().mean(), step=idx)**

**loss\_history.append(loss\_value.numpy().mean()) *# append the loss to the loss\_history record***

**plotter.plot(loss\_history.get())**

***# Backpropagation***

**'''TODO: Use the tape to compute the gradient against all parameters in the CNN model.**

**Use cnn\_model.trainable\_variables to access these parameters.'''**

**grads = *# TODO***

**optimizer.apply\_gradients(zip(grads, cnn\_model.trainable\_variables))**

**comet\_model\_3.log\_figure(figure=plt)**

**comet\_model\_3.end()**

* 1. **Now create a new DNN by reusing and freezing the hidden layers of DNN A and adding a softmax output layer on top with 10 neurons. Train this network on split #2 and see if you can achieve high performance despite having only 500 images per class.**

**reset\_graph()**

**n\_inputs = 28 \* 28 *# MNIST***

**n\_hidden1 = 300 *# reused***

**n\_hidden2 = 50 *# reused***

**n\_hidden3 = 50 *# reused***

**n\_hidden4 = 20 *# new!***

**n\_outputs = 10 *# new!***

**X = tf.placeholder(tf.float32, shape=(None, n\_inputs), name="X")**

**y = tf.placeholder(tf.int32, shape=(None), name="y")**

**with tf.name\_scope("dnn"):**

**hidden1 = tf.layers.dense(X, n\_hidden1, activation=tf.nn.relu, name="hidden1") *# reused***

**hidden2 = tf.layers.dense(hidden1, n\_hidden2, activation=tf.nn.relu, name="hidden2") *# reused***

**hidden3 = tf.layers.dense(hidden2, n\_hidden3, activation=tf.nn.relu, name="hidden3") *# reused***

**hidden4 = tf.layers.dense(hidden3, n\_hidden4, activation=tf.nn.relu, name="hidden4") *# new!***

**logits = tf.layers.dense(hidden4, n\_outputs, name="outputs") *# new!***

**with tf.name\_scope("loss"):**

**xentropy = tf.nn.sparse\_softmax\_cross\_entropy\_with\_logits(labels=y, logits=logits)**

**loss = tf.reduce\_mean(xentropy, name="loss")**

**with tf.name\_scope("eval"):**

**correct = tf.nn.in\_top\_k(logits, y, 1)**

**accuracy = tf.reduce\_mean(tf.cast(correct, tf.float32), name="accuracy")**