

Homework 4 - Machine Learning Training Pipeline - Spring 2025

Essay Question: *Machine learning (ML) pipelines offer many benefits to the ML engineer responsible for designing, training, and evaluating ML models that perform well without overfitting the available data. These benefits are ...* List what you see as the benefits of these pipelines, for each benefit include a sentence describing how that benefit helps the process of model development. Conclude with a sentence summarizing the critical role that such pipelines play in developing ML applications.

Notebook Assignment 1 (Naive Bayes Baseline Model) :

Introduction: A widely used baseline model for text classification is the Naive Bayes model. The text classification problem has input samples $\mathbf{x} \in \{0, 1\}^d$ that are binary vectors of dimension d . The i th component, x_i , represents the i th word in a predefined vocabulary list and $x_i = 1$ if the i th word in the vocabulary is present in a text document and is 0 otherwise. The target, $y \in \{0, 1\}$, for input sample \mathbf{x} is 1 if the text document is in class 1 (let's say a paper about deep learning) otherwise it is 0 (a document that is about something other than deep learning). The text classification problem uses the dataset to train a model that predicts which class a document belongs to. In general, we would classify the document, \mathbf{x} , with the class that has the highest posterior probability $\Pr(y | \mathbf{x})$. We will use our Naive Bayes' baseline model to compute an estimate, $\widehat{\Pr}(y | \mathbf{x})$, of that posterior probability, which we would then use to compute the baseline accuracy levels we need our deep learning model to beat.

Our estimate for $\widehat{\Pr}(y | \mathbf{x})$ is obtained by first using Bayes' theorem to note that

$$\Pr(y | \mathbf{x}) = \frac{\Pr(\mathbf{x} | y)\Pr(y)}{\Pr(\mathbf{x})} \propto \Pr(\mathbf{x} | y)\Pr(y)$$

The probability $\Pr(y)$ can be estimated from the dataset by simply counting up how many samples are in class $y = 1$ or class $y = 0$ and dividing by the total number of samples in the dataset.

We need to estimate the a priori probability $\Pr(\mathbf{x} | y)$, which is what our "trained" model should do. But to get a crude baseline model for this, we can simply assume that \mathbf{x} is a random variable whose components, x_i , are independent Bernoulli random variables

$$\Pr(x_i | y) = x_i \Pr(x_i = 1 | y) + (1 - x_i) \Pr(x_i = 0 | y)$$

The baseline model's estimate of the document's a priori probability is therefore,

$$\Pr(\mathbf{x} | y) = \prod_{i=1}^d (x_i \Pr(x_i = 1 | y) + (1 - x_i) \Pr(x_i = 0 | y))$$

The probability $\Pr(x_i = 1 | y)$ can be estimated from the dataset by simply counting up how many times word i appears in documents that are of class y .

So we can now propose the following procedure to generate a Naive Bayes' baseline model for the text classification problem

1. Define vocabulary list V where the cardinality of V determines the dimension of the input vectors, \mathbf{x} .

2. Count the following in the dataset

- N = total number of documents in the dataset
- N_m = number of documents labelled with class m for $m = 1, 2, \dots, M$
- $n_m(x_i)$ = number of documents of class k containing word i

3. Estimate the likelihoods $\widehat{\text{Pr}}(x_i | y = m) = \frac{n_m(x_i)}{N_m}$

4. Estimate the priors $\widehat{\text{Pr}}(y = m) = \frac{N_m}{N}$

5. Estimate the baseline model's prediction of the posterior *log likelihood* for each class

$$\begin{aligned} \log(\widehat{\text{Pr}}(y = m | \mathbf{x})) &= \log(\widehat{\text{Pr}}(\mathbf{x} | y = m)) + \log(\text{Pr}(y = m)) \\ &= \log(\widehat{\text{Pr}}(y = m)) + \sum_{i=1}^{|V|} \log \left(x_i \widehat{\text{Pr}}(x_i | y = m) + (1 - x_i)(1 - \widehat{\text{Pr}}(x_i | y = m)) \right) \end{aligned}$$

6. The baseline model's prediction is to select the class that has the largest log likelihood. One then evaluates the model's performance (accuracy or confusion matrix) on the test data.

1. **Exercise 1:** Consider a set of document, each of which is related to *Sports* (S) or to *Informatics* (I). Define a vocabulary of eight words,

$$V = \begin{bmatrix} x_1 = \text{goal} \\ x_2 = \text{tutor} \\ x_3 = \text{variance} \\ x_4 = \text{speed} \\ x_5 = \text{drink} \\ x_6 = \text{defence} \\ x_7 = \text{performance} \\ x_8 = \text{field} \end{bmatrix}$$

So each document is an 8-dimensional binary vector. Let the input samples be

$$\left\{ \begin{array}{cccccc|cccc} 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & 0 & 0 & 0 \\ \hline s & s & s & s & s & s & i & i & i & i & i \end{array} \right\}$$

where each column's first 8 elements are the inputs for that sample and the last element is the class label (s or i). Classify the following input sample using the Naive Bayes Classifier

$$\mathbf{x} = [1 \ 0 \ 0 \ 1 \ 1 \ 1 \ 0 \ 1]^T$$

Solution: The total number of documents in the training set is $N = 11$ with $N_{\text{sport}} = 6$ and $N_{\text{inf}} = 5$. The prior probabilities are therefore

$$\widehat{\text{Pr}}(\text{sport}) = \frac{6}{11}, \quad \widehat{\text{Pr}}(\text{inf}) = \frac{5}{11}$$

The document count, $n_{\text{sport}}(x_i)$ and $n_{\text{inf}}(x_i)$ are obtained for each component of the feature inputs

	\mathbf{x}	$n_{\text{sport}}(x_i)$	$\widehat{P}(x_i \text{sport})$	$n_{\text{inf}}(x_i)$	$\widehat{P}(x_i \text{inf})$
x_1	1	3	3/6	1	1/5
x_2	0	1	1/6	3	3/5
x_3	0	2	2/6	3	3/5
x_4	1	3	3/6	1	1/5
x_5	1	3	3/6	1	1/5
x_6	1	4	4/6	1	1/5
x_7	0	4	4/6	3	3/5
x_8	1	4	4/6	1	1/5

Table 1: table

We then compute the posterior loglikelihood of the class as

$$\begin{aligned} \log(\widehat{\text{Pr}}(\text{sport} | \mathbf{x})) &\propto \log \widehat{\text{Pr}}(\text{sport}) \sum_{i=1}^8 \log \left(x_i \widehat{\text{Pr}}(x_i | \text{sport}) + (1 - x_i)(1 - \widehat{\text{Pr}}(x_i | \text{sport})) \right) \\ &\propto \log\left(\frac{6}{11}\right) + \log \left(\frac{3}{6} \times \frac{5}{6} \times \frac{2}{3} \times \frac{1}{2} \times \frac{2}{3} \times \frac{1}{3} \times \frac{2}{3} \right) = -4.4898 \\ \widehat{\text{Pr}}(\text{inf} | \mathbf{x}) &\propto \widehat{\text{Pr}}(\text{inf}) \prod_{i=1}^8 \left(x_i \widehat{\text{Pr}}(x_i | \text{inf}) + (1 - x_i)(1 - \widehat{\text{Pr}}(x_i | \text{inf})) \right) \\ &\propto \frac{5}{11} \left(\frac{1}{5} \times \frac{2}{5} \times \frac{2}{5} \times \frac{1}{5} \times \frac{1}{5} \times \frac{1}{5} \times \frac{2}{5} \times \frac{1}{5} \right) = -9.9751 \end{aligned}$$

Since $-4.4898 = \log \widehat{\text{Pr}}(\text{sport} | \log \mathbf{x}) > \widehat{\text{Pr}}(\text{inf} | \mathbf{x}) = -9.9751$ we classify this document as sport.

- Exercise 2:** Consider the IMDB dataset whose input samples, \mathbf{x} , are encoded as binary vectors of dimension 10,000 with binary valued targets. Use the IMDB training data to determine a Naive Bayes' baseline model. Evaluate the baseline model's accuracy on the IMDB testing data.

Solution: We first load the imdb dataset

```
from tensorflow.keras.datasets import imdb
import numpy as np
num_words = 10000
(train_x, train_y), (test_x, test_y) = imdb.load_data(num_words=num_words)
```

We then take the input samples of the training and test sets and encode them as one-hot vectors. The scripts for doing this are in the optimizer section of chapter 4.

```

import numpy as npd
def encode(sequences, dimension = 10000):
    results = np.zeros((len(sequences),dimension))
    for i, sequence in enumerate(sequences):
        for j in sequence:
            results[i, j] = 1.
    return results

train_x = encode(train_x)
test_x = encode(test_x)
train_y = np.asarray(train_y).astype("float32")
test_y = np.asarray(test_y).astype("float32")

```

I broke the development of the Naive Bayes baseline model into two functions. The first function `baseline_model` computes the dataset statistics $\widehat{\Pr}(y = m)$ and $\widehat{\Pr}(x_i | y = m)$ for each word in the vocabulary.

```

def baseline_model(samples, labels):
    N = len(labels)
    indx1 = np.where(labels==1)[0]
    indx0 = np.where(labels==0)[0]
    N1 = len(indx1)
    X1 = samples[indx1,:]
    N0 = len(indx0)
    X0 = samples[indx0,:]
    n1 = np.sum(X1,axis=0)
    n0 = np.sum(X0,axis=0)
    P1 = N1/N
    P0 = N0/N
    P1x = n1/N1
    P0x = n0/N0
    return [P1,P0,P1x,P0x]

model = baseline_model(train_x, train_y)

```

This function takes the training dataset and returns the four probabilities in a list called `model`. We then use that model to evaluate the performance (i.e. accuracy) of the baseline model's predictions on the testing dataset.

```

def baseline_evaluate(model, samples, labels):
    #samples = train_x
    #labels = train_y
    P1 = model[0]
    P0 = model[1]
    P1x = model[2]
    P0x = model[3]

    error = np.zeros(len(samples))
    for entry, sample in enumerate(samples):
        label = labels[entry]
        #print(label)
        post1 = P1*(sample*P1x+(1-sample)*(1-P1x))
        post0 = P0*(sample*P0x+(1-sample)*(1-P0x))

        logP1 = np.sum(np.log(post1))
        logP0 = np.sum(np.log(post0))

        if logP1>logP0:
            pred=1.
        else:
            pred=0.

        #print([pred, label])

        if (pred!=label):
            error[entry]=1

    error_rate = np.mean(error)
    accuracy = 1- error_rate
    return accuracy

baseline_accuracy = baseline_evaluate(model, test_x, test_y)
print(f"baseline test accuracy = {100*baseline_accuracy:.2f}%")

```

Note that we use the posterior log-likelihood, rather than likelihood. This is because if we used the likelihood, the repeated product of probabilities would be very "small". The outcomes establishes that the baseline model's test accuracy is 83.90%.

3. Now instantiate a neural network model whose input layer takes the 10000 dimensional input sample and maps it through two dense layers of 4 nodes each using an relu activation function. The output layer of the model has a single node with a sigmoid activation function. Compile your model using an RMSprop optimizer with a learning rate of 10^{-4} , binary crossentropy loss function, and "accuracy" as the metric. Train the model for 60 epochs using 80% of the data samples in the original imdb training data and using the remaining 20% for validation. Use minibatch training with a batch size of 256. Plot the training curves of your model (training/validation loss and accuracy). Let the "best" model obtained during training be the one with the smallest validation loss and evaluate the "test accuracy" of that model. Compare your result to that obtained with the Bayes Naive baseline model.

Solution: I started this by creating dataset objects for the ptraining, validation, and testing data.

```
#first convert the original IMDB data into training and testing dataset objects
import tensorflow as tf
batch_size = 256
train_ds = tf.data.Dataset.from_tensor_slices((train_x,train_y))
train_ds = train_ds.batch(batch_size)
train_ds_size = len(list(train_ds))
test_ds = tf.data.Dataset.from_tensor_slices((test_x,test_y))
test_ds = test_ds.batch(batch_size)
test_ds_size = len(list(test_ds))
```

We then split `train_ds` into a pre-training dataset and validation dataset object using a 20% split.

```
val_split = 0.20
train_ds_size = len(list(train_ds))
val_size = int(val_split*train_ds_size)
ptrain_size = train_ds_size-val_size
ptrain_ds = train_ds.take(ptrain_size)
val_ds = train_ds.skip(ptrain_size).take(val_size)
```

We now go ahead to instantiate and compile the model

```
from tensorflow import keras
from tensorflow.keras import layers

num_nodes = 4
num_epochs = 60
learning_rate = 1e-4

inputs = keras.Input(shape=(10000,))
x = layers.Dense(num_nodes, activation="relu")(inputs)
x = layers.Dense(num_nodes, activation = "relu")(x)
outputs = layers.Dense(1, activation="sigmoid")(x)
model = keras.Model(inputs=inputs, outputs=outputs)
model.summary

model.compile(
    optimizer = tf.keras.optimizers.RMSprop(learning_rate=learning_rate),
    loss = "binary_crossentropy",
    metrics = ["accuracy"])
```

We now train the model for 60 epochs on the ptraining dataset and use the validation dataset's loss to trigger a callback that saves the "best" model

```
callbacks = [
    keras.callbacks.ModelCheckpoint(
        filepath="test_model.keras",
        save_best_only = True,
        monitor="val_loss"
    )
]

history = model.fit(ptrain_ds, epochs=num_epochs,
    validation_data = val_ds,
    callbacks = callbacks)
```

We then generate the training curves, evaluate the "best" model's accuracy on the testing data and compare to the baseline model's accuracy on the testing data.

```
test_model = keras.models.load_model("test_model.keras")
best_test_loss, best_test_acc = test_model.evaluate(test_ds)

import matplotlib.pyplot as plt
train_loss = history.history["loss"]
val_loss = history.history["val_loss"]
train_acc = history.history["accuracy"]
val_acc = history.history["val_accuracy"]

epochs = range(1, len(train_loss) + 1)

figure, axis = plt.subplots(1,2)
axis[0].plot(epochs, train_loss, "b--", label = "Training loss")
axis[0].plot(epochs, val_loss, "b", label = "Validation loss")
axis[0].set_title(f"Baseline Test Accuracy: {baseline_accuracy: .2f}")
axis[0].legend()
axis[1].plot(epochs, train_acc, "b--", label = "Training Accuracy")
axis[1].plot(epochs, val_acc, "b", label = "Validation Accuracy")
axis[1].set_title(f"Best Test Accuracy: {100*best_test_acc: .2f}%")
axis[1].legend()
```

These results show that the baseline's accuracy is 84% whereas the trained model's accuracy is 87%. So what we see is that the Naive Bayes baseline actually does very well, thereby providing a good starting point for seeing whether our neural network model is actually learning anything useful.

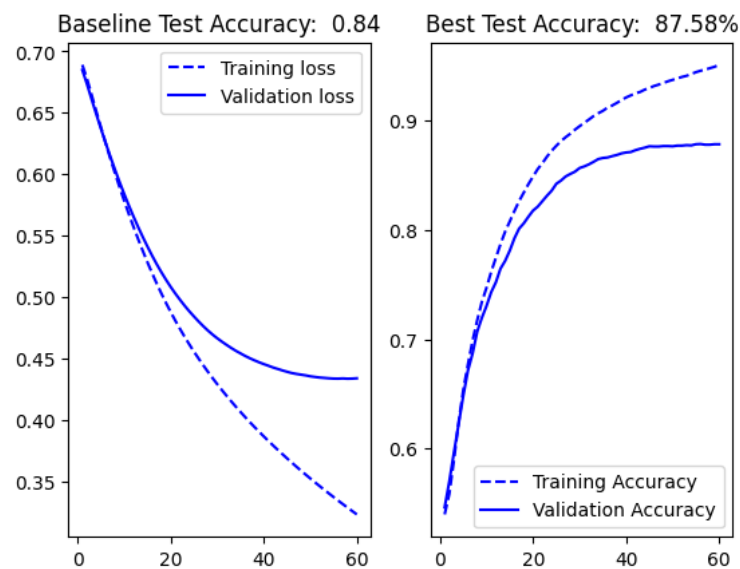


Figure 1: Problem 1 result

Notebook Assignment 2 (Hyperband Tuner): One of the most time-consuming tasks in deep learning is hyperparameter tuning. These hyper-parameters include the number of layers and number of nodes in a deep network. It may include optimizer parameters such as the learning rate or momentum term. It may also include parameters used by a dropout layer or L_2 regularization. Hyperparameter tuning is usually done by creating a finite grid of model configurations, evaluating the loss or accuracy achieved by training a model with the given hyper-parameters, and then using the outcomes to select a new set of parameterized model configurations. The selection criterion is

chosen to "reduce" the number of configurations until the recursive application of this base process identifies a single parameter set that trains a model with the smallest loss (highest accuracy).

Two methods for hyperparameter tuning are Bayesian optimization and the Hyperband algorithm. Both methods have been implemented in the KerasTuner tool which is available in TensorFlow. This notebook assignment has you code a simple tuning algorithm whose basic recursion is the foundation of the Hyperband algorithm. We then have you walk through the same search using TensorFlow's Keras Tuner. You will be working with the fashion MNIST classification problem from HW3.

1. Consider a two layer dense sequential model. The first layer has 128 nodes with RELU activation, followed by a 10 node dense layer. We will treat the number of nodes in that first dense layer as the hyperparameter, `n_hidden`. Load the fashion-mnist dataset and create a numpy array, `hp_bag` which is the set of hyperparameters we want to test.

```
import numpy as np
hp_bag = np.arange(16, 320, 32)
```

`hp_bag` is therefore a collection of `n_hidden` hyper-parameters that we search through in the next exercise for the "best" model using the Hyperband algorithm

2. The hyperband algorithm [1] is based on the idea that you start by training a collection (bag) of models for a few epochs, say 5, and then discard from the bag those models that performed worst. You then continue training for another 5 epochs, discard the worst ones, and continue until there is only one model left, which you then train to completion.

In this first problem train a simple model for the Fashion-MNIST dataset. The model has two dense layers where the first layer has `n_hidden` nodes using an ReLU activation and a 10 node dense layer. The number of nodes in the first hidden layer will be the hyper-parameter we are tuning. To do this, you first create a numpy array, `hp_bag` which contains the set of hyperparameters (i.e. `n_hidden`) we want to test.

```
import numpy as np
hp_bag = np.arange(16, 320, 32)
```

You will write a script that trains the configurations in `hp_bag` for just 5 epochs on the p-training data you split from the Fashion-MNIST training data in HW 2. You will evaluate the loss achieved by all fo the trained models on the validation data and then remove at most `num_remove=3` configurations with the largest losses from `hp_bag`. Your algorithm will repeat this process until there is only one model left in the bag. You will then evaluate generate the training curves for this last model when training for 30 epochs and determine the best model's loss and accuracy.

Note that unless you suppress TensorFlow's logging function, your script will produce a great deal of logging output. You can use the following script to adjust the level of TensorFlow's autoamted logging functions.

```
import os
import logging
def set_tf_loglevel(level):
    if level >= logging.FATAL:
        os.environ['TF_CPP_MIN_LOG_LEVEL']='3'
```

```

if level >=logging.ERROR:
    os.environ['TF_CPP_MIN_LOG_LEVEL']='2'
if level >=logging.WARNING:
    os.environ['TF_CPP_MIN_LOG_LEVEL']='1'
else:
    os.environ['TF_CPP_MIN_LOG_LEVEL']='0'
logging.getLogger('tensorflow').setLevel(level)

set_tf_loglevel(logging.FATAL)

```

Solution: I started by generating the ptraining, validation, and testing datasets from the Fashion-MNIST dataset. This is similar to what you did in HW2.

```

import numpy as np
import matplotlib.pyplot as plt
import tensorflow.keras as keras
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import datasets

(x_train, y_train), (x_test, y_test) = datasets.fashion_mnist.load_data()

x_train_n = np.array(x_train).astype("float32")/255
x_train_n = x_train_n.reshape(60000,28*28)
N,_ = x_train_n.shape
train_split = 2/3
Nptrain = np.ceil(N*train_split).astype("uint32")
x_ptrain = x_train_n[:Nptrain]
y_ptrain = y_train[:Nptrain]
x_val = x_train_n[Nptrain:]
y_val = y_train[Nptrain:]

x_test_n = np.array(x_test).astype("float32")/255
x_test_n = x_test_n.reshape(10000,28*28)

```

I think defined a function that built a model using a given set `hp` of hyperparameters. For this problem there is only one hyperparameter; the number of nodes in the hidden layer.

```

from tensorflow.keras import layers

def build_model(hp):
    model = keras.Sequential([
        keras.Input(shape=(784,)),
        layers.Dense(hp,activation="relu"),
        layers.Dense(10,activation="softmax")
    ])

    model.compile(optimizer='rmsprop',
                  loss = "sparse_categorical_crossentropy",
                  metrics = ['accuracy'])

    return model

def initialize_models(hp_bag):

    for hp in hp_bag:
        model_name = "data/model_"+str(hp)+".keras"

        model = build_model(hp)

        #you need to fit the model
        model.fit(x_ptrain,y_ptrain,epochs = 0, verbose=0)

        model.save(model_name)

        #model.summary()

```

The following script is the main loop I used for the hyperband recursion.

```

from keras.models import load_model

hp_bag = np.arange(16, 320, 32)

print("INITIALIZING MODELS")
initialize_models(hp_bag)

batch = 0
while len(hp_bag)>1:
    print("BATCH "+str(batch))
    result_bag = np.empty((0,3))
    for hp in hp_bag:

```



```

model_name = "data/model_"+str(hp)+".keras"

model = load_model(model_name)

model.fit(x_ptrain,y_ptrain,
          epochs=5,batch_size=512,verbose=0,
          validation_data = (x_val,y_val))

result = model.evaluate(x_val,y_val,verbose=0)

model.save(model_name)

loss   = result[0]
acc    = result[1]
entry  = np.array((hp,loss,acc))
result_bag = np.vstack((result_bag,entry))

batch += 1

print(result_bag)

num_remove = 3
for _ in range(num_remove):
    active_configs = np.shape(hp_bag)
    if active_configs[0] > 1:
        indx = np.argmax(result_bag[:,1])
        result_bag = np.delete(result_bag,indx,0)
        hp_bag=np.delete(hp_bag,indx,0)
print(hp_bag)

```

which produced the output for each BATCH

```

INITIALIZING MODELS
BATCH 0
/Users/michaellemmon/miniconda3/envs/tf/lib/python3.11/site-packages/keras/src/saving/saving_lib.py:576: UserWarning: Skipping variable loading
saveable.load_own_variables(weights_store.get(inner_path))
[[ 16.      0.5241344  0.82209998]
 [ 48.      0.54471242  0.79079998]
 [ 80.      0.46926987  0.83539999]
 [112.      0.50595856  0.82824999]
 [144.      0.51651382  0.8136      ]
 [176.      0.46490663  0.83450001]
 [208.      0.49769789  0.81435001]
 [240.      0.45981461  0.83405     ]
 [272.      0.44445401  0.84044999]
 [304.      0.48474991  0.82795     ]]
[ 80 112 176 208 240 272 304]
BATCH 1
[[ 80.      0.53647697  0.80620003]
 [112.      0.40749145  0.8549      ]
 [176.      0.45521241  0.83060002]
 [208.      0.41527501  0.84560001]
 [240.      0.4102712  0.85000002]
 [272.      0.43135807  0.84364998]
 [304.      0.36038336  0.86914998]]
[112 208 240 304]
BATCH 1
[[112.      0.44476479  0.84434998]
 [208.      0.34902436  0.8732      ]
 [240.      0.34777969  0.87015003]
 [304.      0.33136478  0.88069999]]
[304]

```

The best model was one with 304 nodes in the hidden layer. So we then generated the training curves

```

hp = hp_bag[0]

model = build_model(hp)

model.summary()

callbacks = [
    keras.callbacks.ModelCheckpoint(
        filepath="NA-4-1.keras",
        save_best_only = True,
        monitor = "val_loss"
    )
]

history = model.fit(x_ptrain,
                    y_ptrain,
                    epochs=30,
                    batch_size=512,
                    validation_data = (x_val, y_val),
                    callbacks = callbacks)

```

```
import matplotlib.pyplot as plt
history_dict = history.history
loss_values = history_dict["loss"]
val_loss_values = history_dict["val_loss"]
epochs = range(1, len(loss_values)+1)

test_model = keras.models.load_model("NA-4-1.keras")
test_loss, test_acc = test_model.evaluate(x_test_n, y_test, verbose=1)
print(f"Test accuracy: {test_acc: .3f}")

training_curves(history)
```

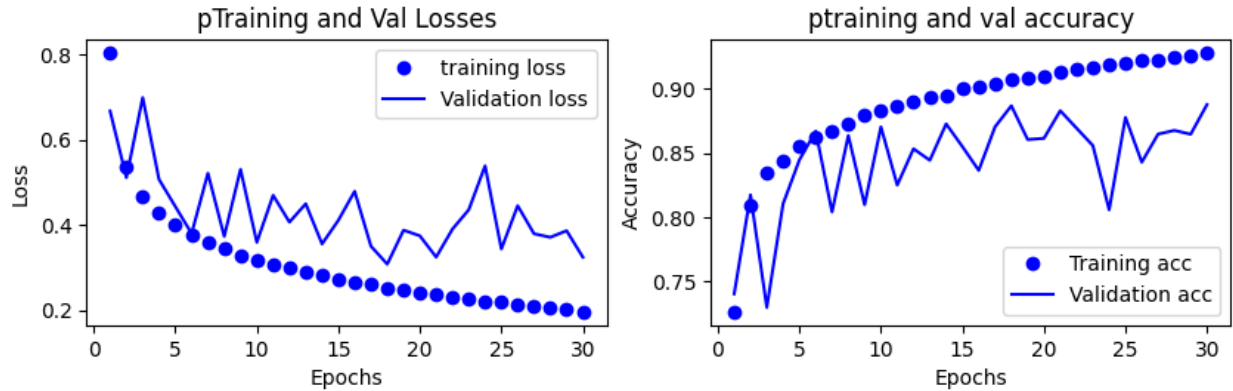


Figure 2: HW4 problem 2 result

- TensorFlow has an automated tuning function based on the Hyperband algorithm described above [2]. TensorFlow's implementation of the algorithm is much more efficient than the script you probably wrote and it is useful to know how to use it. You can make use of the tool in the following script. This script can also be found on the TensorFlow website for `kt_tuner`.

```
import tensorflow as tf
from tensorflow import keras
import keras_tuner as kt

(img_train, label_train), (img_test, label_test) = keras.datasets.fashion_mnist.load_data()
# Normalize pixel values between 0 and 1
img_train = img_train.astype('float32') / 255.0
img_test = img_test.astype('float32') / 255.0

def model_builder(hp):
    model = keras.Sequential()
    model.add(keras.layers.Flatten(input_shape=(28, 28)))

    # Tune the number of units in the first Dense layer
    # Choose an optimal value between 32-512
    hp_units = hp.Int('units', min_value=32, max_value=512, step=32)
    model.add(keras.layers.Dense(units=hp_units, activation='relu'))
    model.add(keras.layers.Dense(10))

    # Tune the learning rate for the optimizer
    # Choose an optimal value from 0.01, 0.001, or 0.0001
    hp_learning_rate = hp.Choice('learning_rate', values=[1e-2, 1e-3, 1e-4])

    model.compile(optimizer=keras.optimizers.Adam(learning_rate=hp_learning_rate),
                  loss=keras.losses.SparseCategoricalCrossentropy(from_logits=True),
                  metrics=['accuracy'])

    return model

tuner = kt.Hyperband(model_builder,
                    objective='val_accuracy',
                    max_epochs=10,
                    factor=3,
                    directory='my_dir',
                    project_name='intro_to_kt')

stop_early = tf.keras.callbacks.EarlyStopping(monitor='val_loss', patience=5)
```

```

tuner.search(img_train, label_train, epochs=50, validation_split=0.2, callbacks=[stop_early])

# Get the optimal hyperparameters
best_hps=tuner.get_best_hyperparameters(num_trials=1)[0]

print(f"""
The hyperparameter search is complete. The optimal number of units in the first densely-connected
layer is {best_hps.get('units')} and the optimal learning rate for the optimizer
is {best_hps.get('learning_rate')}.
""")

```

The output from this identifies the best model after 30 trials as having 512 nodes with a validation accuracy of 89%

```

Trial 30 Complete [00h 00m 17s]
val_accuracy: 0.862333357334137

Best val_accuracy So Far: 0.8924166560173035
Total elapsed time: 00h 04m 05s

The hyperparameter search is complete. The optimal number of units in the first densely-connected
layer is 512 and the optimal learning rate for the optimizer
is 0.001.

```

References

- [1] Lisha Li, Kevin Jamieson, Giulia DeSalvo, Afshin Rostamizadeh, and Ameet Talwalkar. Hyperband: A novel bandit-based approach to hyperparameter optimization. *The Journal of Machine Learning Research*, 18(1):6765–6816, 2017.
- [2] Tom O’Malley, Elie Bursztein, James Long, François Chollet, Haifeng Jin, Luca Invernizzi, et al. Kerastuner. <https://github.com/keras-team/keras-tuner>, 2019.