Evaluation

Learn how to evaluate a pre-trained model stored in a checkpoint.

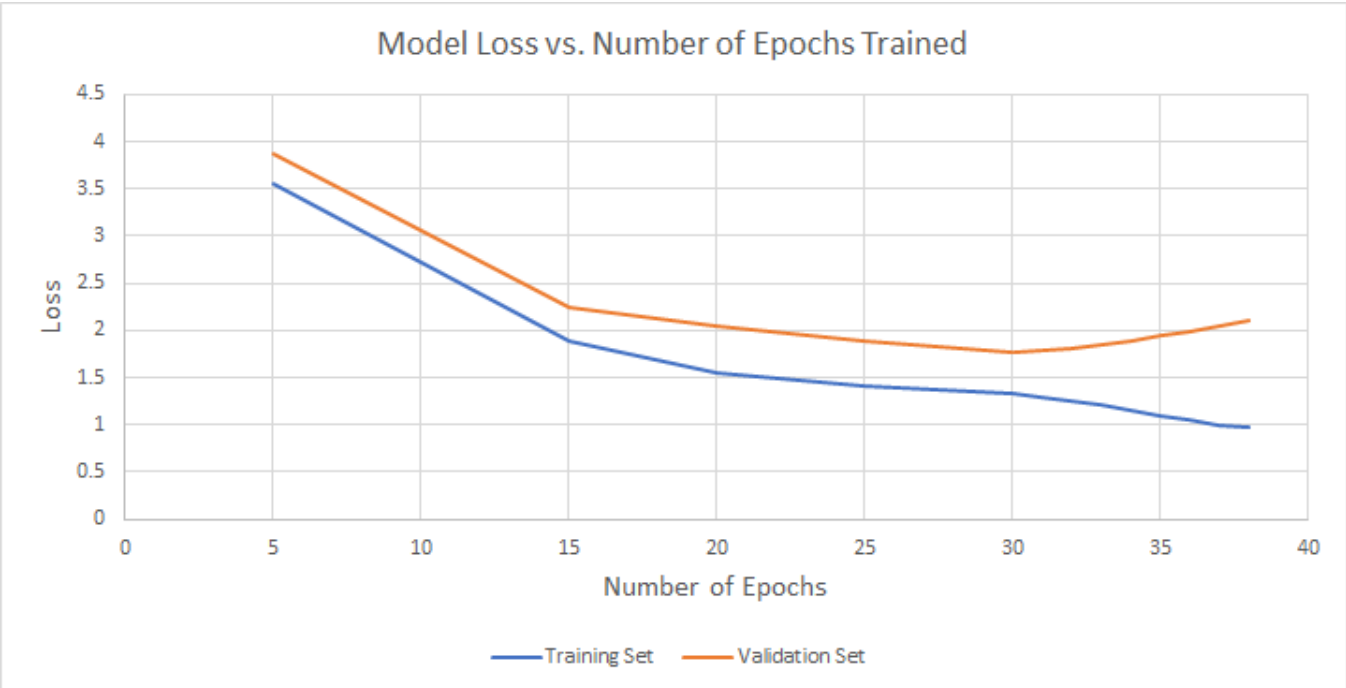
Chapter Goals:

- Understand the difference between training, validation, and test sets
- Learn how to evaluate a trained machine learning model

A. Training vs. evaluation

To measure how well our model has been trained, we evaluate it on datasets other than the training set. The datasets used for evaluation are known as the *validation* and *test* sets. Note that we don't shuffle or repeat the evaluation datasets, since those are techniques used specifically to improve training.

The validation set is used to evaluate a model in between training runs. We use the validation set to tweak certain hyperparameters for a model (such as learning rate or batch size) in order to make sure training continues smoothly. We also use the validation set to detect model overfitting, so we can stop training if overfitting is detected.



Example of overfitting a model on the training set. Overfitting is a common occurrence when training a complex model for too long.

training a complex model for too long.

Overfitting occurs when we train a model (usually a relatively complex model) for too long on the training set, resulting in a decreased ability to generalize well to other datasets. In the above plot, overfitting occurs around the 32nd epoch of training, when the loss on the validation set begins to increase.

The test set is used to evaluate the final version of a model, after it is completely done training. Evaluating on the test set lets us know how well our model performs on its given task.

B. Evaluation metrics

There are a variety of different metrics that can be used for evaluation, depending on the application that a model is built for. However, a universal evaluation metric for machine learning models is the loss. Since every machine learning model is trained to minimize some loss metric, it is natural to use that loss metric during evaluation.

Another commonly used evaluation metric for classification models is accuracy. This refers to the fraction of dataset observations that a machine learning model can label with the correct class. While we train a classification model by minimizing the loss (normally cross entropy), the true goal is to increase model accuracy when classifying new data.

Time to Code!

In this chapter you'll be completing the `evaluate_saved_model` function, which restores a classification model from a checkpoint and then runs evaluation on the model.

First, we get the checkpoint state from the checkpoint directory and make sure the checkpoint file is there.

Set `ckpt` equal to `tf.train.get_checkpoint_state` applied with `ckpt_dir` as the only argument.
Then create an `if` block that checks if `ckpt` is not `None`.

If the checkpoint state is correct, we can set up a `Saver` object and restore the model parameters.

Inside the `if` block, set `saver` equal to a `tf.train.Saver` object, initialized with no arguments.
Then call `saver.restore` with `sess` and `ckpt.model_checkpoint_path` as the two input arguments.

The evaluation metrics we return are the model accuracy and loss. We'll use the input argument `sess` (a `tf.Session` object) to extract the metrics.

Inside the `if` block, set `eval_metrics` equal to `sess.run` with the tuple `(self.accuracy, self.loss)` as the only argument.
Then return `eval_metrics`.

```
import numpy as np
import tensorflow as tf

class ClassificationModel(object):
    def __init__(self, output_size):
        self.output_size = output_size

    # Run model evaluation
    def evaluate_saved_model(self, sess, ckpt_dir):
        # CODE HERE
        pass

    # See the "Efficient Data Processing Techniques" section for details
    def dataset_from_numpy(self, input_data, batch_size, labels=None, is_training=True, num_e
        dataset_input = input_data if labels is None else (input_data, labels)
        dataset = tf.data.Dataset.from_tensor_slices(dataset_input)
        if is_training:
            dataset = dataset.shuffle(len(input_data)).repeat(num_epochs)
        return dataset.batch(batch_size)

    # See the "Machine Learning for Software Engineers" course on Educative
    def run_model_setup(self, inputs, labels, hidden_layers, is_training, calculate_accuracy=
        layer = inputs
        for num_nodes in hidden_layers:
            layer = tf.layers.dense(layer, num_nodes,
                activation=tf.nn.relu)
        logits = tf.layers.dense(layer, self.output_size,
            name='logits')
        self.probs = tf.nn.softmax(logits, name='probs')
        self.predictions = tf.argmax(
            self.probs, axis=-1, name='predictions')
        if calculate_accuracy:
            class_labels = tf.argmax(labels, axis=-1)
            is_correct = tf.equal(
                self.predictions, class_labels)
            is_correct_float = tf.cast(
                is_correct,
                tf.float32)
            self.accuracy = tf.reduce_mean(
                is_correct_float)
```

```
        if labels is not None:
            labels_float = tf.cast(
                labels, tf.float32)
            cross_entropy = tf.nn.softmax_cross_entropy_with_logits_v2(
                labels=labels_float,
                logits=logits)
            self.loss = tf.reduce_mean(
                cross_entropy)
        if is_training:
            adam = tf.train.AdamOptimizer()
            self.train_op = adam.minimize(
                self.loss, global_step=self.global_step)

    # Run training of the classification model
    def run_model_training(self, input_data, labels, hidden_layers, batch_size, num_epochs, c
        self.global_step = tf.train.get_or_create_global_step()
        dataset = self.dataset_from_numpy(input_data, batch_size,
            labels=labels, num_epochs=num_epochs)
        iterator = dataset.make_one_shot_iterator()
        inputs, labels = iterator.get_next()
        self.run_model_setup(inputs, labels, hidden_layers, True)
        tf.summary.scalar('accuracy', self.accuracy)
        tf.summary.histogram('inputs', inputs)
        log_vals = {'loss': self.loss, 'step': self.global_step}
        logging_hook = tf.train.LoggingTensorHook(
            log_vals, every_n_iter=1000)
        nan_hook = tf.train.NanTensorHook(self.loss)
        hooks = [nan_hook, logging_hook]
        with tf.train.MonitoredTrainingSession(
            checkpoint_dir=ckpt_dir,
            hooks=hooks) as sess:
            while not sess.should_stop():
                sess.run(self.train_op)
```