```
# Dense layer
class Layer_Dense:
    ...
    # Retrieve layer parameters
    def get_parameters(self):
        return self.weights, self.biases
```

Within the `Model` class, we'll add `get_parameters` method, which will iterate over the trainable layers of the model, run their `get_parameters` method, and append returned weights and biases to a list:

```
# Model class
class Model:
    ...
    # Retrieves and returns parameters of trainable layers
    def get_parameters(self):

        # Create a list for parameters
        parameters = []

        # Iterable trainable layers and get their parameters
        for layer in self.trainable_layers:
            parameters.append(layer.get_parameters())

        # Return a list
        return parameters
```

Now, after training a model, we can grab the parameters by running:

```
parameters = model.get_parameters()
```

For example:

```
# Create dataset
X, y, X_test, y_test = create_data_mnist('fashion_mnist_images')

# Shuffle the training dataset
keys = np.array(range(X.shape[0]))
np.random.shuffle(keys)
X = X[keys]
y = y[keys]

# Scale and reshape samples
X = (X.reshape(X.shape[0], -1).astype(np.float32) - 127.5) / 127.5
X_test = (X_test.reshape(X_test.shape[0], -1).astype(np.float32) -
             127.5) / 127.5

# Instantiate the model
model = Model()

# Add layers
model.add(Layer_Dense(X.shape[1], 128))
model.add(Activation_ReLU())
model.add(Layer_Dense(128, 128))
model.add(Activation_ReLU())
model.add(Layer_Dense(128, 10))
model.add(Activation_Softmax())

# Set loss, optimizer and accuracy objects
model.set(
    loss=Loss_CategoricalCrossentropy(),
    optimizer=Optimizer_Adam(decay=1e-3),
    accuracy=Accuracy_Categorical()
)

# Finalize the model
model.finalize()

# Train the model
model.train(X, y, validation_data=(X_test, y_test),
            epochs=10, batch_size=128, print_every=100)

# Retrieve and print parameters
parameters = model.get_parameters()
print(parameters)
```

This will look *something* like (we trim the output to save space):

```
[(array([[ 0.03538642,  0.00794717, -0.04143231, ...,  0.04267325,
        -0.00935107,  0.01872394],
       [ 0.03289384,  0.00691249, -0.03424096, ...,  0.02362755,
        -0.00903602,  0.00977725],
       [ 0.02189022, -0.01362374, -0.01442819, ...,  0.01320345,
        -0.02083327,  0.02499157],
       ...,
       [ 0.0146937 , -0.02869027, -0.02198809, ...,  0.01459295,
        -0.02335824,  0.00935643],
       [-0.00090149,  0.01082182, -0.06013806, ...,  0.00704454,
        -0.0039093 ,  0.00311571],
       [ 0.03660082, -0.00809607, -0.02737131, ...,  0.02216582,
        -0.01710589,  0.01578414]], dtype=float32), array([[-2.24505737e-02,
5.40090213e-03,  2.91307438e-02,
        -1.04323691e-02, -9.52822249e-03, -1.48109728e-02,
       ...,
         0.04158591, -0.01614098, -0.0134403 ,  0.00708392,  0.0284729 ,
         0.00336277, -0.00085383,  0.00163819]], dtype=float32)),
(array([[-0.00196577, -0.00335329, -0.01362851, ...,  0.00397028,
         0.00027816,  0.00427755],
       [ 0.04438829, -0.09197803,  0.02897452, ..., -0.11920264,
         0.03808296, -0.00536136],
       [ 0.04146343, -0.03637529,  0.04973305, ..., -0.13564698,
        -0.08259197, -0.02467288],
       ...,
       [ 0.03495856,  0.03902597,  0.0028984 , ..., -0.10016892,
        -0.11356542,  0.05866433],
       [-0.00857899, -0.02612676, -0.01050871, ..., -0.00551328,
        -0.01432311, -0.00916382],
       [-0.20444085, -0.01483698, -0.09321352, ...,  0.02114356,
        -0.0762504 ,  0.03600615]], dtype=float32), array([[-0.0103433 ,
-0.00158314,  0.02268587, -0.02352985, -0.02144126,
        -0.00777614,  0.00795028, -0.00622872,  0.06918745, -0.00743477]],
      dtype=float32))]
```

Setting Parameters

If we have a method to get parameters, we will likely also want to have a method that will set parameters. We'll do this similar to how we setup the `get_parameters` method, starting with the `Layer_Dense` class:

```
# Dense layer
class Layer_Dense:
    ...
    # Set weights and biases in a layer instance
    def set_parameters(self, weights, biases):
        self.weights = weights
        self.biases = biases
```

Then we can update the `Model` class:

```
# Model class
class Model:
    ...
    # Updates the model with new parameters
    def set_parameters(self, parameters):

        # Iterate over the parameters and layers
        # and update each layers with each set of the parameters
        for parameter_set, layer in zip(parameters,
                                        self.trainable_layers):
            layer.set_parameters(*parameter_set)
```

We are also iterating over the trainable layers here, but what we are doing next needs a bit more explanation. First, the `zip()` function takes in iterables, like lists, and returns a new iterable with pairwise combinations of all the iterables passed in as parameters. In other words (and using our example), `zip()` takes a list of parameters and a list of layers and returns an iterator containing tuples of 0th elements of both lists, then the 1st elements of both lists, the 2nd elements from both lists, and so on. This way, we can iterate over parameters and the layer they belong to at the same time. As our parameters are a tuple of weights and biases, we will unpack them with a starred expression so that our `Layer_Dense` method can take them as separate parameters. This approach gives us flexibility if we'd like to use layers with different numbers of parameter groups.

One difference that presents itself now is that this allows us to have a model that never needed an optimizer. If we don't train a model but, instead, load already trained parameters into it, we won't optimize anything. To account for this, we'll visit the `finalize` method of the `Model` class, changing:

```
# Model class
class Model:
    ...
    # Finalize the model
    def finalize(self):
        ...
            # Update loss object with trainable layers
            self.loss.remember_trainable_layers(
                self.trainable_layers
            )
```

To (we added an `if` statement to set a list of trainable layers to the loss function, only if this loss object exists):

```
# Model class
class Model:
    ...
    # Finalize the model
    def finalize(self):
        ...
            # Update loss object with trainable layers
            if self.loss is not None:
                self.loss.remember_trainable_layers(
                    self.trainable_layers
                )
```

Next, we'll change the `Model` class' `set` method to allow us to pass in only given parameters. We'll assign default values and add `if` statements to use parameters only when they're present. To do that, we'll change:

```
    # Set loss, optimizer and accuracy
    def set(self, *, loss, optimizer, accuracy):
        self.loss = loss
        self.optimizer = optimizer
        self.accuracy = accuracy
```

To:

```
    # Set loss, optimizer and accuracy
    def set(self, *, loss=None, optimizer=None, accuracy=None):

        if loss is not None:
            self.loss = loss

        if optimizer is not None:
            self.optimizer = optimizer

        if accuracy is not None:
            self.accuracy = accuracy
```

We can now train a model, retrieve its parameters, create a new model, and set its parameters with those retrieved from the previously-trained model:

```
# Create dataset
X, y, X_test, y_test = create_data_mnist('fashion_mnist_images')

# Shuffle the training dataset
keys = np.array(range(X.shape[0]))
np.random.shuffle(keys)
X = X[keys]
y = y[keys]

# Scale and reshape samples
X = (X.reshape(X.shape[0], -1).astype(np.float32) - 127.5) / 127.5
X_test = (X_test.reshape(X_test.shape[0], -1).astype(np.float32) -
             127.5) / 127.5

# Instantiate the model
model = Model()

# Add layers
model.add(Layer_Dense(X.shape[1], 128))
model.add(Activation_ReLU())
model.add(Layer_Dense(128, 128))
model.add(Activation_ReLU())
model.add(Layer_Dense(128, 10))
model.add(Activation_Softmax())

# Set loss, optimizer and accuracy objects
model.set(
    loss=Loss_CategoricalCrossentropy(),
    optimizer=Optimizer_Adam(decay=1e-4),
    accuracy=Accuracy_Categorical()
)
```

```
# Finalize the model
model.finalize()

# Train the model
model.train(X, y, validation_data=(X_test, y_test),
            epochs=10, batch_size=128, print_every=100)

# Retrieve model parameters
parameters = model.get_parameters()

# New model

# Instantiate the model
model = Model()

# Add layers
model.add(Layer_Dense(X.shape[1], 128))
model.add(Activation_ReLU())
model.add(Layer_Dense(128, 128))
model.add(Activation_ReLU())
model.add(Layer_Dense(128, 10))
model.add(Activation_Softmax())

# Set loss and accuracy objects
# We do not set optimizer object this time - there's no need to do it
# as we won't train the model
model.set(
    loss=Loss_CategoricalCrossentropy(),
    accuracy=Accuracy_Categorical()
)

# Finalize the model
model.finalize()

# Set model with parameters instead of training it
model.set_parameters(parameters)

# Evaluate the model
model.evaluate(X_test, y_test)


>>>
(model training output removed)
validation, acc: 0.874, loss: 0.354
validation, acc: 0.874, loss: 0.354
```

Saving Parameters

We'll extend this further now by actually saving the parameters into a file. To do this, we'll add a `save_parameters` method in the `Model` class. We'll use Python's built-in *pickle* module to serialize any Python object. Serialization is a process of turning an object, which can be of any abstract form, into a binary representation — a set of bytes that can be, for example, saved into a file. This serialized form contains all the information needed to recreate the object later. *Pickle* can either return the bytes of the serialized object or save them directly to a file. We'll make use of the latter ability, so let's import *pickle*:

```
import pickle
```

Then we'll add a new method to the `Model` class. Before having *pickle* save our parameters into a file, we'll need to create a file-handler by opening a file in binary-write mode. We will then pass this handler along to the data into `pickle.dump()`. To create the file, we need a filename that we'll save the data into; we'll pass it in as a parameter:

```
# Model class
class Model:
    ...
    # Saves the parameters to a file
    def save_parameters(self, path):

        # Open a file in the binary-write mode
        # and save parameters to it
        with open(path, 'wb') as f:
            pickle.dump(self.get_parameters(), f)
```

With this method, you can save the parameters of a trained model by running:

```
model.save_parameters('fashion_mnist.parms')
```

Loading Parameters

Presumably, if we are saving model parameters into a file, we would also like to have a way to load them from this file. Loading parameters is very similar to saving the parameters, just reversed. We'll open the file in a binary-read mode and have *pickle* read from it, deserializing parameters back into a list. Then we call the `set_parameters` method that we created earlier and pass in the loaded parameters:

```
    # Loads the weights and updates a model instance with them
    def load_parameters(self, path):

        # Open file in the binary-read mode,
        # load weights and update trainable layers
        with open(path, 'rb') as f:
            self.set_parameters(pickle.load(f))
```

We set up a model, load in the parameters file (we did not train this model), and test the model to check if it works:

```
# Create dataset
X, y, X_test, y_test = create_data_mnist('fashion_mnist_images')

# Shuffle the training dataset
keys = np.array(range(X.shape[0]))
np.random.shuffle(keys)
X = X[keys]
y = y[keys]

# Scale and reshape samples
X = (X.reshape(X.shape[0], -1).astype(np.float32) - 127.5) / 127.5
X_test = (X_test.reshape(X_test.shape[0], -1).astype(np.float32) -
             127.5) / 127.5

# Instantiate the model
model = Model()
```

```
# Add layers
model.add(Layer_Dense(X.shape[1], 128))
model.add(Activation_ReLU())
model.add(Layer_Dense(128, 128))
model.add(Activation_ReLU())
model.add(Layer_Dense(128, 10))
model.add(Activation_Softmax())

# Set loss and accuracy objects
# We do not set optimizer object this time - there's no need to do it
# as we won't train the model
model.set(
    loss=Loss_CategoricalCrossentropy(),
    accuracy=Accuracy_Categorical()
)

# Finalize the model
model.finalize()

# Set model with parameters instead of training it
model.load_parameters('fashion_mnist.parms')

# Evaluate the model
model.evaluate(X_test, y_test)


>>>
validation, acc: 0.874, loss: 0.354
```

While we can save and load model parameter values, we still need to define the model. It must be the exact configuration as the model that we're importing parameters from. It would be easier if we could save the model itself.

Saving the Model

Why didn't we save the whole model in the first place? Saving just weights versus saving the whole model has different use cases along with pros and cons. With saved weights, you can, for example, initialize a model with those weights, trained from similar data, and then train that model to work with your specific data. This is called **transfer learning** and is outside of the scope of this book. Weights can be used to visualize the model (like in some animations that we have created for the purpose of this book, starting from chapter 6), identify dead neurons, implement more complicated models (like **reinforcement learning**, where weights collected from multiple models are committed to a single network), and so on. A file containing just weights is also much smaller than an entire model. A model initialized from weights loads faster and uses less memory, as the optimizer and related parts are not created. One downside of loading just weights and biases is that the initialized model does not contain the optimizer's state. It is possible to train the model further, but it's more optimal to load a full model if we intend to train it. When saving the full model, everything related to it is saved as well; this includes the optimizer's state (that allows us to easily continue the training) and model's structure.

We'll create another method in the `Model` class that we'll use to save the entire model. The first thing we'll do is make a copy of the model since we're going to edit it before saving, and we may also want to save a model during the training process as a **checkpoint**.

```
    # Saves the model
    def save(self, path):

        # Make a deep copy of current model instance
        model = copy.deepcopy(self)
```

We import the *copy* module to support this:

```
import copy
```

The *copy* module offers two methods that allow us to copy the model — *copy* and *deepcopy*. While *copy* is faster, it only copies the first level of the object's properties, causing copies of our model objects to have some references common to the original model. For example, our model object has a list of layers — the list is the top-level property, and the layers themselves

are secondary — therefore, references to the layer objects will be shared by both the original and copied model objects. Due to these challenges with *copy*, we'll use the *deepcopy* method to recursively traverse all objects and create a full copy.

Next, we'll remove the accumulated loss and accuracy:

```
# Reset accumulated values in loss and accuracy objects
model.loss.new_pass()
model.accuracy.new_pass()
```

Then remove any data in the input layer, and reset the gradients, if any exist:

```
# Remove data from input layer
# Remove data from input layer
# and gradients from the loss object
model.input_layer.__dict__.pop('output', None)
model.loss.__dict__.pop('dinputs', None)
```

Both `model.input_layer` and `model.loss` are class instances. They're attributes of the `Model` object but also objects themselves. One of the dunder properties (called "dunder" because of the double underscores) that exists for all classes is the `__dict__` property. It contains names and values for the class object's properties. We can then use the built-in `pop` method on these values, which means we remove them from that instance of the class' object. The `pop` method will wind up throwing an error if the key we pass as the first parameter doesn't exist, as the `pop` method wants to return the value of the key that it removes. We use the second parameter of the `pop` method — which is the default value that we want to return if the key doesn't exist — to prevent these errors. We will set this parameter to *None* — we do not intend to catch the removed values, and it doesn't really matter what the default value is. This way, we do not have to check if a given property exists, in times like when we'd like to delete it using the *del* statement, and some of them might not exist.

Next, we'll iterate over all the layers to remove their properties:

```
# For each layer remove inputs, output and dinputs properties
for layer in model.layers:
    for property in ['inputs', 'output', 'dinputs',
                     'dweights', 'dbiases']:
        layer.__dict__.pop(property, None)
```

With these things cleaned up, we can save the model object. To do that, we have to open a file in a binary-write mode, and call `pickle.dump()` with the model object and the file handler as parameters:

```
# Open a file in the binary-write mode and save the model
with open(path, 'wb') as f:
    pickle.dump(model, f)
```

This makes the full `save` method:

```
    # Saves the model
    def save(self, path):

        # Make a deep copy of current model instance
        model = copy.deepcopy(self)

        # Reset accumulated values in loss and accuracy objects
        model.loss.new_pass()
        model.accuracy.new_pass()

        # Remove data from the input layer
        # and gradients from the loss object
        model.input_layer.__dict__.pop('output', None)
        model.loss.__dict__.pop('dinputs', None)

        # For each layer remove inputs, output and dinputs properties
        for layer in model.layers:
            for property in ['inputs', 'output', 'dinputs',
                             'dweights', 'dbiases']:
                layer.__dict__.pop(property, None)

        # Open a file in the binary-write mode and save the model
        with open(path, 'wb') as f:
            pickle.dump(model, f)
```

This means we can train a model, then save it whenever we wish with:

```
model.save('fashion_mnist.model')
```

Loading the Model

Loading a model will ideally take place before a model object even exists. What we mean by this is we could load a model by calling a method of the `Model` class instead of the object:

```
model = Model.load('fashion_mnist.model')
```

To achieve this, we're going to use the `@staticmethod` decorator. This decorator can be used with class methods to run them on uninitialized objects, where the *`self`* does not exist (notice that it is missing the function definition). In our case, we're going to use it to immediately create a model object without first needing to instantiate a model object. Within this method, we'll open a file using the passed-in path, in binary-read mode, and use pickle to deserialize the saved model:

```
    # Loads and returns a model
    @staticmethod
    def load(path):

        # Open file in the binary-read mode, load a model
        with open(path, 'rb') as f:
            model = pickle.load(f)

        # Return a model
        return model
```

Since we already have a saved model, let's create the data, and then load a model to see if it works:

```
# Create dataset
X, y, X_test, y_test = create_data_mnist('fashion_mnist_images')

# Shuffle the training dataset
keys = np.array(range(X.shape[0]))
np.random.shuffle(keys)
X = X[keys]
y = y[keys]
```

```
# Scale and reshape samples
X = (X.reshape(X.shape[0], -1).astype(np.float32) - 127.5) / 127.5
X_test = (X_test.reshape(X_test.shape[0], -1).astype(np.float32) -
             127.5) / 127.5

# Load the model
model = Model.load('fashion_mnist.model')

# Evaluate the model
model.evaluate(X_test, y_test)


>>>
validation, acc: 0.874, loss: 0.354
```

Saving the full trained model is a common way of saving a model. It saves parameters (weights and biases) and instances of all the model's objects and the data they generated. That is going to be, for example, the optimizer state like cache, learning rate decay, full model structure, etc. Loading the model, in this case, is as easy as calling one method and the model is ready to use, whether we want to continue training it or use it for a prediction.

Supplementary Material: https://nnfs.io/ch21
Chapter code, further resources, and errata for this chapter.

Chapter 22

Prediction / Inference

While we often spend most of our time focusing on training and testing models, the whole reason we're doing any of this is to have a model that takes new inputs and produces desired outputs. This will typically involve many attempts to train the best model possible, save that model, and load that saved model to do inference, or prediction.

In the case of Fashion MNIST classification, we'd like to load a trained model, show it never-before-seen images, and have it predict the correct classification. To do this, we'll add a new `predict` method to the `Model` class:

```
# Predicts on the samples
def predict(self, X, *, batch_size=None):
```

Note that we predict *`X`* with a possible *`batch_size`*. This means all predictions, including predictions on just one sample, will still be fed in as a list of samples in the form of a NumPy array, whose first dimension is the list samples, and second is sample data. For example, if we would like to predict on a single image, we still need to create a NumPy array mimicking a list containing a single sample — with a shape of *(1, 784)* where *1* is this single sample, and *784* is

the number of features in a sample (pixels per image). Similar to the `evaluate` method, we'll calculate the number of steps we plan to take:

```
        # Default value if batch size is not being set
        prediction_steps = 1

        # Calculate number of steps
        if batch_size is not None:
            prediction_steps = len(X) // batch_size
            # Dividing rounds down. If there are some remaining
            # data, but not a full batch, this won't include it
            # Add `1` to include this not full batch
            if prediction_steps * batch_size < len(X):
                prediction_steps += 1
```

Then create a list that we'll populate with the predictions:

```
        # Model outputs
        output = []
```

We'll iterate over the batches, passing the samples for predictions forward through the network, and populating the `output` with the predictions:

```
        # Iterate over steps
        for step in range(prediction_steps):

            # If batch size is not set -
            # train using one step and full dataset
            if batch_size is None:
                batch_X = X

            # Otherwise slice a batch
            else:
                batch_X = X[step*batch_size:(step+1)*batch_size]

            # Perform the forward pass
            batch_output = self.forward(batch_X, training=False)

            # Append batch prediction to the list of predictions
            output.append(batch_output)
```

After running this, the `output` is a list of batch predictions. Each of them is a NumPy array, a partial result made by predicting on a batch of samples from the input data array. Any applications, or programs, that will make use of the inference output of our models, we expect to simply pass in a list of samples and get back a list of predictions (both in the form of a NumPy array as mentioned before). Since we're not focused on training, we're only using batches in prediction

to ensure our model can fit into memory, but we're going to get a return that's also in batches of predictions. We can see a simple example of this:

```
import numpy as np

output = []

b = np.array([[1, 2], [3, 4]])
output.append(b)
b = np.array([[5, 6], [7, 8]])
output.append(b)
b = np.array([[9, 10], [11, 12]])
output.append(b)

print(output)

>>>
[array([[1, 2],
       [3, 4]]), array([[5, 6],
       [7, 8]]), array([[ 9, 10],
       [11, 12]])]
```

In this example, we see an output with a batch size of 2 and 6 total samples. The output is a list of arrays, with each array housing a batch of predictions. Instead, we want just 1 list of predictions, no more batches. To achieve this, we're going to use NumPy's `vstack` method:

```
import numpy as np

output = []

b = np.array([[1, 2], [3, 4]])
output.append(b)
b = np.array([[5, 6], [7, 8]])
output.append(b)
b = np.array([[9, 10], [11, 12]])
output.append(b)

output = np.vstack(output)

print(output)

>>>
[[ 1  2]
 [ 3  4]
 [ 5  6]
 [ 7  8]
 [ 9 10]
 [11 12]]
```

It takes a list of objects and stacks them, if possible, creating a homologous array. This is a preferable form of the return from the `predict` method when we pass a list of samples. With plain Python, we might just add to the list each step:

```
output = []

b = [[1, 2], [3, 4]]
output += b
b = [[5, 6], [7, 8]]
output += b
b = [[9, 10], [11, 12]]
output += b

print(output)


>>>
[[1, 2], [3, 4], [5, 6], [7, 8], [9, 10], [11, 12]]
```

We add results to a list and stack them at the end, instead of appending to the NumPy array each batch to avoid a performance penalty. Unlike plain Python, NumPy is written in *C* language and creates data objects in memory differently. That means that there is no easy way of adding data to the existing NumPy array, other than merging two arrays and saving the result as a new array. But this will lead to a performance penalty, since the further in predictions we are, the bigger the resulting array is. The fastest and most optimal way is to append NumPy arrays to a list and stack them vertically at once when we have collected all of the partial results. We'll add the `np.vstack` to the end of the outputs that we return:

```
        # Stack and return results
        return np.vstack(output)
```

Making our full `predict` method:

```
    # Predicts on the samples
    def predict(self, X, *, batch_size=None):

        # Default value if batch size is not being set
        prediction_steps = 1

        # Calculate number of steps
        if batch_size is not None:
            prediction_steps = len(X) // batch_size
            # Dividing rounds down. If there are some remaining
            # data, but not a full batch, this won't include it
            # Add `1` to include this not full batch
            if prediction_steps * batch_size < len(X):
                prediction_steps += 1

        # Model outputs
        output = []

        # Iterate over steps
        for step in range(prediction_steps):

            # If batch size is not set -
            # train using one step and full dataset
            if batch_size is None:
                batch_X = X

            # Otherwise slice a batch
            else:
                batch_X = X[step*batch_size:(step+1)*batch_size]

            # Perform the forward pass
            batch_output = self.forward(batch_X, training=False)

            # Append batch prediction to the list of predictions
            output.append(batch_output)

        # Stack and return results
        return np.vstack(output)
```

Now we can load the model and test the prediction functionality:

```
# Create dataset
X, y, X_test, y_test = create_data_mnist('fashion_mnist_images')

# Scale and reshape samples
X_test = (X_test.reshape(X_test.shape[0], -1).astype(np.float32) -
             127.5) / 127.5
```

```
# Load the model
model = Model.load('fashion_mnist.model')

# Predict on the first 5 samples from validation dataset
# and print the result
confidences = model.predict(X_test[:5])
print(confidences)
```

```
>>>
[[9.6826810e-01 8.3330568e-05 1.0794386e-03 1.3643305e-03 7.6704117e-07
  5.5963554e-08 2.9197156e-02 8.6661328e-16 6.8134182e-06 1.8056496e-12]
 [7.7293724e-01 2.0613789e-03 9.3451981e-04 9.0647154e-02 3.4899445e-04
  2.0565639e-07 1.3301854e-01 6.3095896e-12 5.2045987e-05 7.7830048e-11]
 [9.4310820e-01 5.1831361e-05 1.4724518e-03 8.1068231e-04 7.9751426e-06
  9.9619001e-07 5.4532889e-02 2.9622423e-13 1.4997837e-05 2.2963499e-10]
 [9.8930722e-01 1.2575739e-04 2.5738587e-04 1.4423713e-04 2.5113836e-06
  5.6183376e-07 1.0156924e-02 2.8593078e-13 5.5162018e-06 1.4746830e-10]
 [9.2869467e-01 7.3713978e-04 1.7579789e-03 2.1864739e-03 1.7945129e-05
  1.9282908e-05 6.6521421e-02 5.1533548e-11 6.5157568e-05 7.2020221e-09]]
```

It looks like it's working! After spending so much time training and finding the best hyperparameters, a common issue people have is actually *using* the model. As a reminder, each of the subarrays in the output is a vector of confidences containing a confidence metric per class. The first thing that we need to do in this case is to gather the argmax values of these confidence vectors. Recall that we're using a softmax classifier, so this neural network is attempting to fit to one-hot vectors, where the correct class is represented by a 1, and the others by 0s. When doing inference, it is unlikely to achieve such a perfect result, but the index associated with the highest value in the output is what we determine the model is predicting; we're just using the argmax. We could write code to do this, but we've already done that in all of the activation function classes, where we added a `predictions` method:

```
# Softmax activation
class Activation_Softmax:
    ...
    # Calculate predictions for outputs
    def predictions(self, outputs):
        return np.argmax(outputs, axis=1)
```

We've also set an attribute in our model with the output layer's activation function, which means we can generically acquire predictions by performing:

```
# Load the model
model = Model.load('fashion_mnist.model')

# Predict on the first 5 samples from validation dataset and print the
result
confidences = model.predict(X_test[:5])
predictions = model.output_layer_activation.predictions(confidences)
print(predictions)

# Print first 5 labels
print(y_test[:5])

>>>
[0 0 0 0 0]
[0 0 0 0 0]
```

In this case, our model predicted all "class 0," and our test labels were all class 0 as well. Since shuffling our testing data isn't essential, we never shuffled them, so they're going in the original order like our training data was. This explains why all these predictions are 0s.

In practice, we don't care what class number something is, we want to know *what* it is. In this case, class numbers map directly to names, so we add the following dictionary to our code:

```
fashion_mnist_labels = {
    0: 'T-shirt/top',
    1: 'Trouser',
    2: 'Pullover',
    3: 'Dress',
    4: 'Coat',
    5: 'Sandal',
    6: 'Shirt',
    7: 'Sneaker',
    8: 'Bag',
    9: 'Ankle boot'
}
```

Then we could get the string classification by performing:

```
for prediction in predictions:
    print(fashion_mnist_labels[prediction])

>>>
T-shirt/top
T-shirt/top
T-shirt/top
T-shirt/top
T-shirt/top
```

This is great, but we still have to *actually* predict something instead of the training data. When covering deep learning, the training steps often get all the focus; we want to see those accuracy and loss metrics look good! It works well to focus on training for tutorials that aim to show people how to *use* a framework, but one of the larger pain points we see is *applying* the models in production, or just running predictions on new data that was sourced from the wild (especially since outside data is rarely formatted to match your training data).

At the moment, we have a model trained on items of clothing, so we need some truly new samples. Luckily, you're probably a person who owns some clothes; if so, you can take photos of those to start with. If not, use the following sample photos:

https://nnfs.io/datasets/tshirt.png

Fig 20.01: Hand-made t-shirt image for the purpose of inference.

https://nnfs.io/datasets/pants.png

Fig 20.02: Hand-made pants image for the purpose of inference.

You can also try your hand at hand-drawing samples like these. Once you have new images/samples that you wish to use in production, you'll need to preprocess them in the same way the training samples were. Some of these changes are fairly difficult to forget, like the image resolution or number of color channels; we'd get an error if we didn't do those things. Let's start preprocessing our image by loading it in. We'll use the *cv2* package to read in the image:

```
import cv2

image_data = cv2.imread('tshirt.png', cv2.IMREAD_UNCHANGED)
```

We can view the image:

```
import matplotlib.pyplot as plt
plt.imshow(cv2.cvtColor(image_data, cv2.COLOR_BGR2RGB))
plt.show()
```

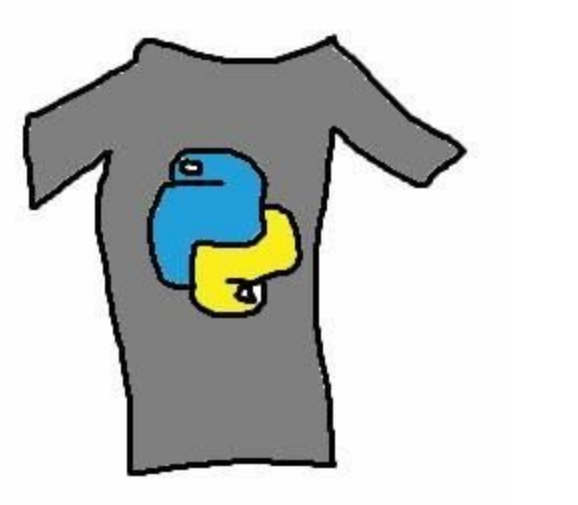

Fig 20.03: Hand-made t-shirt image loaded with Python.

Note that we're doing *cv2.cvtColor* because OpenCV uses BGR (blue, green, red pixel values) color format by default, but matplotlib uses RGB (red, green, blue), so we're converting the colormap to display the image.

The first thing we'll do is read this image as grayscale instead of RGB. This is in contrast to the Fashion MNIST images, which are grayscaled, and we have used `cv2.IMREAD_UNCHANGED` as a parameter to the `cv2.imread()` to inform OpenCV that our intention is to read images grayscaled and unchanged. Here, we have a color image, and this parameter won't work as "unchanged" means containing all the colors; thus, we'll use `cv2.IMREAD_GRAYSCALE` to force grayscaling when we read in our image:

```
import cv2
image_data = cv2.imread('tshirt.png', cv2.IMREAD_GRAYSCALE)
```

Then we can display it:

```
import matplotlib.pyplot as plt
plt.imshow(image_data, cmap='gray')
plt.show()
```

Note that we use a gray colormap with `plt.imshow()` by passing the `'gray'` argument into the *cmap* parameter. The result is a grayscale image:

Fig 20.04: Grayscaled hand-made t-shirt image loaded with Python.

Next, we'll resize the image to be the same 28x28 resolution as our training data:

```
image_data = cv2.resize(image_data, (28, 28))
```

We then display this resized image:

```
plt.imshow(image_data, cmap='gray')
plt.show()
```

Fig 20.05: Grayscaled and scaled down hand-made t-shirt image.

Next, we'll flatten and scale the image. While the scale operation is the same as for the training data, the flattening is a bit different; we don't have a list of images but a single image, and, as previously explained, a single image must be passed in as a list containing this single image. We flatten by applying `.reshape(1, -1)` to the image. The `1` argument represents the number of samples, and the `-1` flattens the image to a vector of length 784. This produces a 1x784 array with our one sample and 784 features (i.e., 28x28 pixels):

```
import numpy as np

image_data = (image_data.reshape(1, -1).astype(np.float32) -
                  127.5) / 127.5
```

Now we can load in our model and predict on this image data:

```
# Load the model
model = Model.load('fashion_mnist.model')

# Predict on the image
confidences = model.predict(image_data)

# Get prediction instead of confidence levels
predictions = model.output_layer_activation.predictions(confidences)

# Get label name from label index
prediction = fashion_mnist_labels[predictions[0]]

print(prediction)
```

Making our code up to this point that loads, preprocesses, and predicts:

```
# Label index to label name relation
fashion_mnist_labels = {
    0: 'T-shirt/top',
    1: 'Trouser',
    2: 'Pullover',
    3: 'Dress',
    4: 'Coat',
    5: 'Sandal',
    6: 'Shirt',
    7: 'Sneaker',
    8: 'Bag',
    9: 'Ankle boot'
}

# Read an image
image_data = cv2.imread('tshirt.png', cv2.IMREAD_GRAYSCALE)
```

```
# Resize to the same size as Fashion MNIST images
image_data = cv2.resize(image_data, (28, 28))

# Reshape and scale pixel data
image_data = (image_data.reshape(1, -1).astype(np.float32) -
                 127.5) / 127.5

# Load the model
model = Model.load('fashion_mnist.model')

# Predict on the image
predictions = model.predict(image_data)

# Get prediction instead of confidence levels
predictions = model.output_layer_activation.predictions(predictions)

# Get label name from label index
prediction = fashion_mnist_labels[predictions[0]]

print(prediction)
```

Note that we are using `predictions[0]` as we passed in a single image in the form of a list, and the model returns a list containing a single prediction.

Only one problem…

```
>>>
Ankle boot
```

What's wrong? Let's compare our currently-preprocessed image to the training data:

```
import matplotlib.pyplot as plt

mnist_image = cv2.imread('fashion_mnist_images/train/0/0000.png',
                         cv2.IMREAD_UNCHANGED)
plt.imshow(mnist_image, cmap='gray')
plt.show()
```

Fig 20.06: Example t-shirt image from the Fashion MNIST dataset

Now we compare this original and example training image to our's:

Fig 20.07: Grayscaled and scaled down hand-made t-shirt image.

The training data that we've used is color-inverted (i.e., the background is black instead of white, and so on). To invert our image before scaling, we can use pixel math directly instead of using OpenCV. We'll subtract all the pixel values from the maximum pixel value: 255. For example, a value of *0* will become *255 - 0 = 255*, and the value of *255* will become *255 - 255 = 0*.

```
image_data = 255 - image_data
```

With this small change, our prediction code becomes:

```
# Read an image
image_data = cv2.imread('tshirt.png', cv2.IMREAD_GRAYSCALE)

# Resize to the same size as Fashion MNIST images
image_data = cv2.resize(image_data, (28, 28))

# Invert image colors
image_data = 255 - image_data
```

```
# Reshape and scale pixel data
image_data = (image_data.reshape(1, -1).astype(np.float32) -
                 127.5) / 127.5

# Load the model
model = Model.load('fashion_mnist.model')

# Predict on the image
confidences = model.predict(image_data)

# Get prediction instead of confidence levels
predictions = model.output_layer_activation.predictions(confidences)

# Get label name from label index
prediction = fashion_mnist_labels[predictions[0]]

print(prediction)


>>>
T-shirt/top
```

Now it works! The reason it works now, and not work previously, is from how the *Dense* layers work — they learn feature (pixel in this case) values and the correlation between them. Contrast this with convolutional layers, which are being trained to find and understand features on images (not features as data input nodes, but actual characteristics/traits, such as lines and curves). Because pixel values were very different, the model incorrectly put its "guess" in this case. Convolutional layers may properly predict in this case, as-is.

Let's try the pants:

```
import matplotlib.pyplot as plt

image_data = cv2.imread('pants.png', cv2.IMREAD_UNCHANGED)
plt.imshow(cv2.cvtColor(image_data, cv2.COLOR_BGR2RGB))
plt.show()
```

Fig 20.08: Hand-made pants image loaded with Python.

Now we'll preprocess:

```
# Read an image
image_data = cv2.imread('pants.png', cv2.IMREAD_GRAYSCALE)

# Resize to the same size as Fashion MNIST images
image_data = cv2.resize(image_data, (28, 28))

# Invert image colors
image_data = 255 - image_data
```

Let's see what we have:

```
plt.imshow(image_data, cmap='gray')
plt.show()
```

Fig 20.09: Grayscaled and scaled down hand-made t-shirt image.

Making our code:

```
# Label index to label name relation
fashion_mnist_labels = {
    0: 'T-shirt/top',
    1: 'Trouser',
    2: 'Pullover',
    3: 'Dress',
    4: 'Coat',
    5: 'Sandal',
    6: 'Shirt',
    7: 'Sneaker',
    8: 'Bag',
    9: 'Ankle boot'
}

# Read an image
image_data = cv2.imread('pants.png', cv2.IMREAD_GRAYSCALE)

# Resize to the same size as Fashion MNIST images
image_data = cv2.resize(image_data, (28, 28))

# Invert image colors
image_data = 255 - image_data

# Reshape and scale pixel data
image_data = (image_data.reshape(1, -1).astype(np.float32) -
                127.5) / 127.5

# Load the model
model = Model.load('fashion_mnist.model')

# Predict on the image
confidences = model.predict(image_data)

# Get prediction instead of confidence levels
predictions = model.output_layer_activation.predictions(confidences)

# Get label name from label index
prediction = fashion_mnist_labels[predictions[0]]

print(prediction)

>>>
Trouser
```

A success again! We have now coded in the last feature of our model, which closes the list of the topics that we covered in this book.

Full code:

```
import numpy as np
import nnfs
import os
import cv2
import pickle
import copy

nnfs.init()


# Dense layer
class Layer_Dense:

    # Layer initialization
    def __init__(self, n_inputs, n_neurons,
                 weight_regularizer_l1=0, weight_regularizer_l2=0,
                 bias_regularizer_l1=0, bias_regularizer_l2=0):
        # Initialize weights and biases
        self.weights = 0.01 * np.random.randn(n_inputs, n_neurons)
        self.biases = np.zeros((1, n_neurons))
        # Set regularization strength
        self.weight_regularizer_l1 = weight_regularizer_l1
        self.weight_regularizer_l2 = weight_regularizer_l2
        self.bias_regularizer_l1 = bias_regularizer_l1
        self.bias_regularizer_l2 = bias_regularizer_l2

    # Forward pass
    def forward(self, inputs, training):
        # Remember input values
        self.inputs = inputs
        # Calculate output values from inputs, weights and biases
        self.output = np.dot(inputs, self.weights) + self.biases

    # Backward pass
    def backward(self, dvalues):
        # Gradients on parameters
        self.dweights = np.dot(self.inputs.T, dvalues)
        self.dbiases = np.sum(dvalues, axis=0, keepdims=True)
```

```
        # Gradients on regularization
        # L1 on weights
        if self.weight_regularizer_l1 > 0:
            dL1 = np.ones_like(self.weights)
            dL1[self.weights < 0] = -1
            self.dweights += self.weight_regularizer_l1 * dL1
        # L2 on weights
        if self.weight_regularizer_l2 > 0:
            self.dweights += 2 * self.weight_regularizer_l2 * \
                             self.weights
        # L1 on biases
        if self.bias_regularizer_l1 > 0:
            dL1 = np.ones_like(self.biases)
            dL1[self.biases < 0] = -1
            self.dbiases += self.bias_regularizer_l1 * dL1
        # L2 on biases
        if self.bias_regularizer_l2 > 0:
            self.dbiases += 2 * self.bias_regularizer_l2 * \
                            self.biases

        # Gradient on values
        self.dinputs = np.dot(dvalues, self.weights.T)

    # Retrieve layer parameters
    def get_parameters(self):
        return self.weights, self.biases

    # Set weights and biases in a layer instance
    def set_parameters(self, weights, biases):
        self.weights = weights
        self.biases = biases


# Dropout
class Layer_Dropout:

    # Init
    def __init__(self, rate):
        # Store rate, we invert it as for example for dropout
        # of 0.1 we need success rate of 0.9
        self.rate = 1 - rate

    # Forward pass
    def forward(self, inputs, training):
        # Save input values
        self.inputs = inputs
```

```
        # If not in the training mode - return values
        if not training:
            self.output = inputs.copy()
            return

        # Generate and save scaled mask
        self.binary_mask = np.random.binomial(1, self.rate,
                           size=inputs.shape) / self.rate
        # Apply mask to output values
        self.output = inputs * self.binary_mask

    # Backward pass
    def backward(self, dvalues):
        # Gradient on values
        self.dinputs = dvalues * self.binary_mask


# Input "layer"
class Layer_Input:

    # Forward pass
    def forward(self, inputs, training):
        self.output = inputs


# ReLU activation
class Activation_ReLU:

    # Forward pass
    def forward(self, inputs, training):
        # Remember input values
        self.inputs = inputs
        # Calculate output values from inputs
        self.output = np.maximum(0, inputs)

    # Backward pass
    def backward(self, dvalues):
        # Since we need to modify original variable,
        # let's make a copy of values first
        self.dinputs = dvalues.copy()

        # Zero gradient where input values were negative
        self.dinputs[self.inputs <= 0] = 0

    # Calculate predictions for outputs
    def predictions(self, outputs):
        return outputs
```

```
# Softmax activation
class Activation_Softmax:

    # Forward pass
    def forward(self, inputs, training):
        # Remember input values
        self.inputs = inputs

        # Get unnormalized probabilities
        exp_values = np.exp(inputs - np.max(inputs, axis=1,
                                            keepdims=True))
        # Normalize them for each sample
        probabilities = exp_values / np.sum(exp_values, axis=1,
                                            keepdims=True)

        self.output = probabilities

    # Backward pass
    def backward(self, dvalues):

        # Create uninitialized array
        self.dinputs = np.empty_like(dvalues)

        # Enumerate outputs and gradients
        for index, (single_output, single_dvalues) in \
                enumerate(zip(self.output, dvalues)):
            # Flatten output array
            single_output = single_output.reshape(-1, 1)
            # Calculate Jacobian matrix of the output and
            jacobian_matrix = np.diagflat(single_output) - \
                              np.dot(single_output, single_output.T)
            # Calculate sample-wise gradient
            # and add it to the array of sample gradients
            self.dinputs[index] = np.dot(jacobian_matrix,
                                         single_dvalues)

    # Calculate predictions for outputs
    def predictions(self, outputs):
        return np.argmax(outputs, axis=1)
```

```
# Sigmoid activation
class Activation_Sigmoid:

    # Forward pass
    def forward(self, inputs, training):
        # Save input and calculate/save output
        # of the sigmoid function
        self.inputs = inputs
        self.output = 1 / (1 + np.exp(-inputs))

    # Backward pass
    def backward(self, dvalues):
        # Derivative - calculates from output of the sigmoid function
        self.dinputs = dvalues * (1 - self.output) * self.output

    # Calculate predictions for outputs
    def predictions(self, outputs):
        return (outputs > 0.5) * 1


# Linear activation
class Activation_Linear:

    # Forward pass
    def forward(self, inputs, training):
        # Just remember values
        self.inputs = inputs
        self.output = inputs

    # Backward pass
    def backward(self, dvalues):
        # derivative is 1, 1 * dvalues = dvalues - the chain rule
        self.dinputs = dvalues.copy()

    # Calculate predictions for outputs
    def predictions(self, outputs):
        return outputs
```

```
# SGD optimizer
class Optimizer_SGD:

    # Initialize optimizer - set settings,
    # learning rate of 1. is default for this optimizer
    def __init__(self, learning_rate=1., decay=0., momentum=0.):
        self.learning_rate = learning_rate
        self.current_learning_rate = learning_rate
        self.decay = decay
        self.iterations = 0
        self.momentum = momentum

    # Call once before any parameter updates
    def pre_update_params(self):
        if self.decay:
            self.current_learning_rate = self.learning_rate * \
                (1. / (1. + self.decay * self.iterations))

    # Update parameters
    def update_params(self, layer):

        # If we use momentum
        if self.momentum:

            # If layer does not contain momentum arrays, create them
            # filled with zeros
            if not hasattr(layer, 'weight_momentums'):
                layer.weight_momentums = np.zeros_like(layer.weights)
                # If there is no momentum array for weights
                # The array doesn't exist for biases yet either.
                layer.bias_momentums = np.zeros_like(layer.biases)

            # Build weight updates with momentum - take previous
            # updates multiplied by retain factor and update with
            # current gradients
            weight_updates = \
                self.momentum * layer.weight_momentums - \
                self.current_learning_rate * layer.dweights
            layer.weight_momentums = weight_updates

            # Build bias updates
            bias_updates = \
                self.momentum * layer.bias_momentums - \
                self.current_learning_rate * layer.dbiases
            layer.bias_momentums = bias_updates
```

```
            # Vanilla SGD updates (as before momentum update)
            else:
                weight_updates = -self.current_learning_rate * \
                                 layer.dweights
                bias_updates = -self.current_learning_rate * \
                               layer.dbiases

            # Update weights and biases using either
            # vanilla or momentum updates
            layer.weights += weight_updates
            layer.biases += bias_updates

        # Call once after any parameter updates
        def post_update_params(self):
            self.iterations += 1


    # Adagrad optimizer
    class Optimizer_Adagrad:

        # Initialize optimizer - set settings
        def __init__(self, learning_rate=1., decay=0., epsilon=1e-7):
            self.learning_rate = learning_rate
            self.current_learning_rate = learning_rate
            self.decay = decay
            self.iterations = 0
            self.epsilon = epsilon

        # Call once before any parameter updates
        def pre_update_params(self):
            if self.decay:
                self.current_learning_rate = self.learning_rate * \
                    (1. / (1. + self.decay * self.iterations))

        # Update parameters
        def update_params(self, layer):

            # If layer does not contain cache arrays,
            # create them filled with zeros
            if not hasattr(layer, 'weight_cache'):
                layer.weight_cache = np.zeros_like(layer.weights)
                layer.bias_cache = np.zeros_like(layer.biases)

            # Update cache with squared current gradients
            layer.weight_cache += layer.dweights**2
            layer.bias_cache += layer.dbiases**2
```

```
            # Vanilla SGD parameter update + normalization
            # with square rooted cache
            layer.weights += -self.current_learning_rate * \
                             layer.dweights / \
                             (np.sqrt(layer.weight_cache) + self.epsilon)
            layer.biases += -self.current_learning_rate * \
                            layer.dbiases / \
                            (np.sqrt(layer.bias_cache) + self.epsilon)

    # Call once after any parameter updates
    def post_update_params(self):
        self.iterations += 1


# RMSprop optimizer
class Optimizer_RMSprop:

    # Initialize optimizer - set settings
    def __init__(self, learning_rate=0.001, decay=0., epsilon=1e-7,
                 rho=0.9):
        self.learning_rate = learning_rate
        self.current_learning_rate = learning_rate
        self.decay = decay
        self.iterations = 0
        self.epsilon = epsilon
        self.rho = rho

    # Call once before any parameter updates
    def pre_update_params(self):
        if self.decay:
            self.current_learning_rate = self.learning_rate * \
                (1. / (1. + self.decay * self.iterations))

    # Update parameters
    def update_params(self, layer):

        # If layer does not contain cache arrays,
        # create them filled with zeros
        if not hasattr(layer, 'weight_cache'):
            layer.weight_cache = np.zeros_like(layer.weights)
            layer.bias_cache = np.zeros_like(layer.biases)

        # Update cache with squared current gradients
        layer.weight_cache = self.rho * layer.weight_cache + \
            (1 - self.rho) * layer.dweights**2
        layer.bias_cache = self.rho * layer.bias_cache + \
            (1 - self.rho) * layer.dbiases**2
```

```
            # Vanilla SGD parameter update + normalization
            # with square rooted cache
            layer.weights += -self.current_learning_rate * \
                             layer.dweights / \
                             (np.sqrt(layer.weight_cache) + self.epsilon)
            layer.biases += -self.current_learning_rate * \
                            layer.dbiases / \
                            (np.sqrt(layer.bias_cache) + self.epsilon)

    # Call once after any parameter updates
    def post_update_params(self):
        self.iterations += 1


# Adam optimizer
class Optimizer_Adam:

    # Initialize optimizer - set settings
    def __init__(self, learning_rate=0.001, decay=0., epsilon=1e-7,
                 beta_1=0.9, beta_2=0.999):
        self.learning_rate = learning_rate
        self.current_learning_rate = learning_rate
        self.decay = decay
        self.iterations = 0
        self.epsilon = epsilon
        self.beta_1 = beta_1
        self.beta_2 = beta_2

    # Call once before any parameter updates
    def pre_update_params(self):
        if self.decay:
            self.current_learning_rate = self.learning_rate * \
                (1. / (1. + self.decay * self.iterations))

    # Update parameters
    def update_params(self, layer):

        # If layer does not contain cache arrays,
        # create them filled with zeros
        if not hasattr(layer, 'weight_cache'):
            layer.weight_momentums = np.zeros_like(layer.weights)
            layer.weight_cache = np.zeros_like(layer.weights)
            layer.bias_momentums = np.zeros_like(layer.biases)
            layer.bias_cache = np.zeros_like(layer.biases)
```

```
        # Update momentum  with current gradients
        layer.weight_momentums = self.beta_1 * \
                                 layer.weight_momentums + \
                                 (1 - self.beta_1) * layer.dweights
        layer.bias_momentums = self.beta_1 * \
                               layer.bias_momentums + \
                               (1 - self.beta_1) * layer.dbiases
        # Get corrected momentum
        # self.iteration is 0 at first pass
        # and we need to start with 1 here
        weight_momentums_corrected = layer.weight_momentums / \
            (1 - self.beta_1 ** (self.iterations + 1))
        bias_momentums_corrected = layer.bias_momentums / \
            (1 - self.beta_1 ** (self.iterations + 1))
        # Update cache with squared current gradients
        layer.weight_cache = self.beta_2 * layer.weight_cache + \
            (1 - self.beta_2) * layer.dweights**2
        layer.bias_cache = self.beta_2 * layer.bias_cache + \
            (1 - self.beta_2) * layer.dbiases**2
        # Get corrected cache
        weight_cache_corrected = layer.weight_cache / \
            (1 - self.beta_2 ** (self.iterations + 1))
        bias_cache_corrected = layer.bias_cache / \
            (1 - self.beta_2 ** (self.iterations + 1))

        # Vanilla SGD parameter update + normalization
        # with square rooted cache
        layer.weights += -self.current_learning_rate * \
                         weight_momentums_corrected / \
                         (np.sqrt(weight_cache_corrected) +
                             self.epsilon)
        layer.biases += -self.current_learning_rate * \
                         bias_momentums_corrected / \
                         (np.sqrt(bias_cache_corrected) +
                             self.epsilon)

    # Call once after any parameter updates
    def post_update_params(self):
        self.iterations += 1
```

```
# Common loss class
class Loss:

    # Regularization loss calculation
    def regularization_loss(self):

        # 0 by default
        regularization_loss = 0

        # Calculate regularization loss
        # iterate all trainable layers
        for layer in self.trainable_layers:

            # L1 regularization - weights
            # calculate only when factor greater than 0
            if layer.weight_regularizer_l1 > 0:
                regularization_loss += layer.weight_regularizer_l1 * \
                                       np.sum(np.abs(layer.weights))

            # L2 regularization - weights
            if layer.weight_regularizer_l2 > 0:
                regularization_loss += layer.weight_regularizer_l2 * \
                                       np.sum(layer.weights * \
                                              layer.weights)

            # L1 regularization - biases
            # calculate only when factor greater than 0
            if layer.bias_regularizer_l1 > 0:
                regularization_loss += layer.bias_regularizer_l1 * \
                                       np.sum(np.abs(layer.biases))

            # L2 regularization - biases
            if layer.bias_regularizer_l2 > 0:
                regularization_loss += layer.bias_regularizer_l2 * \
                                       np.sum(layer.biases * \
                                              layer.biases)

        return regularization_loss

    # Set/remember trainable layers
    def remember_trainable_layers(self, trainable_layers):
        self.trainable_layers = trainable_layers
```

```
    # Calculates the data and regularization losses
    # given model output and ground truth values
    def calculate(self, output, y, *, include_regularization=False):

        # Calculate sample losses
        sample_losses = self.forward(output, y)

        # Calculate mean loss
        data_loss = np.mean(sample_losses)

        # Add accumulated sum of losses and sample count
        self.accumulated_sum += np.sum(sample_losses)
        self.accumulated_count += len(sample_losses)

        # If just data loss - return it
        if not include_regularization:
            return data_loss

        # Return the data and regularization losses
        return data_loss, self.regularization_loss()

    # Calculates accumulated loss
    def calculate_accumulated(self, *, include_regularization=False):

        # Calculate mean loss
        data_loss = self.accumulated_sum / self.accumulated_count

        # If just data loss - return it
        if not include_regularization:
            return data_loss

        # Return the data and regularization losses
        return data_loss, self.regularization_loss()

    # Reset variables for accumulated loss
    def new_pass(self):
        self.accumulated_sum = 0
        self.accumulated_count = 0


# Cross-entropy loss
class Loss_CategoricalCrossentropy(Loss):

    # Forward pass
    def forward(self, y_pred, y_true):

        # Number of samples in a batch
        samples = len(y_pred)
```

```
        # Clip data to prevent division by 0
        # Clip both sides to not drag mean towards any value
        y_pred_clipped = np.clip(y_pred, 1e-7, 1 - 1e-7)

        # Probabilities for target values -
        # only if categorical labels
        if len(y_true.shape) == 1:
            correct_confidences = y_pred_clipped[
                range(samples),
                y_true
            ]

        # Mask values - only for one-hot encoded labels
        elif len(y_true.shape) == 2:
            correct_confidences = np.sum(
                y_pred_clipped * y_true,
                axis=1
            )

        # Losses
        negative_log_likelihoods = -np.log(correct_confidences)
        return negative_log_likelihoods

    # Backward pass
    def backward(self, dvalues, y_true):

        # Number of samples
        samples = len(dvalues)
        # Number of labels in every sample
        # We'll use the first sample to count them
        labels = len(dvalues[0])

        # If labels are sparse, turn them into one-hot vector
        if len(y_true.shape) == 1:
            y_true = np.eye(labels)[y_true]

        # Calculate gradient
        self.dinputs = -y_true / dvalues
        # Normalize gradient
        self.dinputs = self.dinputs / samples
```

```
# Softmax classifier - combined Softmax activation
# and cross-entropy loss for faster backward step
class Activation_Softmax_Loss_CategoricalCrossentropy():

    # Backward pass
    def backward(self, dvalues, y_true):

        # Number of samples
        samples = len(dvalues)

        # If labels are one-hot encoded,
        # turn them into discrete values
        if len(y_true.shape) == 2:
            y_true = np.argmax(y_true, axis=1)

        # Copy so we can safely modify
        self.dinputs = dvalues.copy()
        # Calculate gradient
        self.dinputs[range(samples), y_true] -= 1
        # Normalize gradient
        self.dinputs = self.dinputs / samples


# Binary cross-entropy loss
class Loss_BinaryCrossentropy(Loss):

    # Forward pass
    def forward(self, y_pred, y_true):

        # Clip data to prevent division by 0
        # Clip both sides to not drag mean towards any value
        y_pred_clipped = np.clip(y_pred, 1e-7, 1 - 1e-7)

        # Calculate sample-wise loss
        sample_losses = -(y_true * np.log(y_pred_clipped) +
                          (1 - y_true) * np.log(1 - y_pred_clipped))
        sample_losses = np.mean(sample_losses, axis=-1)

        # Return losses
        return sample_losses

    # Backward pass
    def backward(self, dvalues, y_true):

        # Number of samples
        samples = len(dvalues)
        # Number of outputs in every sample
        # We'll use the first sample to count them
        outputs = len(dvalues[0])
```

```
        # Clip data to prevent division by 0
        # Clip both sides to not drag mean towards any value
        clipped_dvalues = np.clip(dvalues, 1e-7, 1 - 1e-7)

        # Calculate gradient
        self.dinputs = -(y_true / clipped_dvalues -
                         (1 - y_true) / (1 - clipped_dvalues)) / outputs
        # Normalize gradient
        self.dinputs = self.dinputs / samples


# Mean Squared Error loss
class Loss_MeanSquaredError(Loss):  # L2 loss

    # Forward pass
    def forward(self, y_pred, y_true):

        # Calculate loss
        sample_losses = np.mean((y_true - y_pred)**2, axis=-1)

        # Return losses
        return sample_losses

    # Backward pass
    def backward(self, dvalues, y_true):

        # Number of samples
        samples = len(dvalues)
        # Number of outputs in every sample
        # We'll use the first sample to count them
        outputs = len(dvalues[0])

        # Gradient on values
        self.dinputs = -2 * (y_true - dvalues) / outputs
        # Normalize gradient
        self.dinputs = self.dinputs / samples


# Mean Absolute Error loss
class Loss_MeanAbsoluteError(Loss):  # L1 loss

    def forward(self, y_pred, y_true):

        # Calculate loss
        sample_losses = np.mean(np.abs(y_true - y_pred), axis=-1)

        # Return losses
        return sample_losses
```

```
    # Backward pass
    def backward(self, dvalues, y_true):

        # Number of samples
        samples = len(dvalues)
        # Number of outputs in every sample
        # We'll use the first sample to count them
        outputs = len(dvalues[0])

        # Calculate gradient
        self.dinputs = np.sign(y_true - dvalues) / outputs
        # Normalize gradient
        self.dinputs = self.dinputs / samples


# Common accuracy class
class Accuracy:

    # Calculates an accuracy
    # given predictions and ground truth values
    def calculate(self, predictions, y):

        # Get comparison results
        comparisons = self.compare(predictions, y)

        # Calculate an accuracy
        accuracy = np.mean(comparisons)

        # Add accumulated sum of matching values and sample count
        self.accumulated_sum += np.sum(comparisons)
        self.accumulated_count += len(comparisons)

        # Return accuracy
        return accuracy

    # Calculates accumulated accuracy
    def calculate_accumulated(self):

        # Calculate an accuracy
        accuracy = self.accumulated_sum / self.accumulated_count

        # Return the data and regularization losses
        return accuracy

    # Reset variables for accumulated accuracy
    def new_pass(self):
        self.accumulated_sum = 0
        self.accumulated_count = 0
```

```
# Accuracy calculation for classification model
class Accuracy_Categorical(Accuracy):

    # No initialization is needed
    def init(self, y):
        pass

    # Compares predictions to the ground truth values
    def compare(self, predictions, y):
        if len(y.shape) == 2:
            y = np.argmax(y, axis=1)
        return predictions == y


# Accuracy calculation for regression model
class Accuracy_Regression(Accuracy):

    def __init__(self):
        # Create precision property
        self.precision = None

    # Calculates precision value
    # based on passed in ground truth values
    def init(self, y, reinit=False):
        if self.precision is None or reinit:
            self.precision = np.std(y) / 250

    # Compares predictions to the ground truth values
    def compare(self, predictions, y):
        return np.absolute(predictions - y) < self.precision


# Model class
class Model:

    def __init__(self):
        # Create a list of network objects
        self.layers = []
        # Softmax classifier's output object
        self.softmax_classifier_output = None

    # Add objects to the model
    def add(self, layer):
        self.layers.append(layer)
```

```
    # Set loss, optimizer and accuracy
    def set(self, *, loss=None, optimizer=None, accuracy=None):

        if loss is not None:
            self.loss = loss

        if optimizer is not None:
            self.optimizer = optimizer

        if accuracy is not None:
            self.accuracy = accuracy

    # Finalize the model
    def finalize(self):

        # Create and set the input layer
        self.input_layer = Layer_Input()

        # Count all the objects
        layer_count = len(self.layers)

        # Initialize a list containing trainable layers:
        self.trainable_layers = []

        # Iterate the objects
        for i in range(layer_count):

            # If it's the first layer,
            # the previous layer object is the input layer
            if i == 0:
                self.layers[i].prev = self.input_layer
                self.layers[i].next = self.layers[i+1]

            # All layers except for the first and the last
            elif i < layer_count - 1:
                self.layers[i].prev = self.layers[i-1]
                self.layers[i].next = self.layers[i+1]

            # The last layer - the next object is the loss
            # Also let's save aside the reference to the last object
            # whose output is the model's output
            else:
                self.layers[i].prev = self.layers[i-1]
                self.layers[i].next = self.loss
                self.output_layer_activation = self.layers[i]
```

```
            # If layer contains an attribute called "weights",
            # it's a trainable layer -
            # add it to the list of trainable layers
            # We don't need to check for biases -
            # checking for weights is enough
            if hasattr(self.layers[i], 'weights'):
                self.trainable_layers.append(self.layers[i])

            # Update loss object with trainable layers
            if self.loss is not None:
                self.loss.remember_trainable_layers(
                    self.trainable_layers
                )

        # If output activation is Softmax and
        # loss function is Categorical Cross-Entropy
        # create an object of combined activation
        # and loss function containing
        # faster gradient calculation
        if isinstance(self.layers[-1], Activation_Softmax) and \
           isinstance(self.loss, Loss_CategoricalCrossentropy):
            # Create an object of combined activation
            # and loss functions
            self.softmax_classifier_output = \
                Activation_Softmax_Loss_CategoricalCrossentropy()

    # Train the model
    def train(self, X, y, *, epochs=1, batch_size=None,
              print_every=1, validation_data=None):

        # Initialize accuracy object
        self.accuracy.init(y)

        # Default value if batch size is not being set
        train_steps = 1

        # Calculate number of steps
        if batch_size is not None:
            train_steps = len(X) // batch_size
            # Dividing rounds down. If there are some remaining
            # data but not a full batch, this won't include it
            # Add `1` to include this not full batch
            if train_steps * batch_size < len(X):
                train_steps += 1
```

```
        # Main training loop
        for epoch in range(1, epochs+1):

            # Print epoch number
            print(f'epoch: {epoch}')

            # Reset accumulated values in loss and accuracy objects
            self.loss.new_pass()
            self.accuracy.new_pass()

            # Iterate over steps
            for step in range(train_steps):

                # If batch size is not set -
                # train using one step and full dataset
                if batch_size is None:
                    batch_X = X
                    batch_y = y

                # Otherwise slice a batch
                else:
                    batch_X = X[step*batch_size:(step+1)*batch_size]
                    batch_y = y[step*batch_size:(step+1)*batch_size]

                # Perform the forward pass
                output = self.forward(batch_X, training=True)

                # Calculate loss
                data_loss, regularization_loss = \
                    self.loss.calculate(output, batch_y,
                                        include_regularization=True)
                loss = data_loss + regularization_loss

                # Get predictions and calculate an accuracy
                predictions = self.output_layer_activation.predictions(
                                  output)
                accuracy = self.accuracy.calculate(predictions,
                                                   batch_y)

                # Perform backward pass
                self.backward(output, batch_y)

                # Optimize (update parameters)
                self.optimizer.pre_update_params()
                for layer in self.trainable_layers:
                    self.optimizer.update_params(layer)
                self.optimizer.post_update_params()
```

```
            # Print a summary
            if not step % print_every or step == train_steps - 1:
                print(f'step: {step}, ' +
                      f'acc: {accuracy:.3f}, ' +
                      f'loss: {loss:.3f} (' +
                      f'data_loss: {data_loss:.3f}, ' +
                      f'reg_loss: {regularization_loss:.3f}), ' +
                      f'lr: {self.optimizer.current_learning_rate}')

        # Get and print epoch loss and accuracy
        epoch_data_loss, epoch_regularization_loss = \
            self.loss.calculate_accumulated(
                include_regularization=True)
        epoch_loss = epoch_data_loss + epoch_regularization_loss
        epoch_accuracy = self.accuracy.calculate_accumulated()

        print(f'training, ' +
              f'acc: {epoch_accuracy:.3f}, ' +
              f'loss: {epoch_loss:.3f} (' +
              f'data_loss: {epoch_data_loss:.3f}, ' +
              f'reg_loss: {epoch_regularization_loss:.3f}), ' +
              f'lr: {self.optimizer.current_learning_rate}')

        # If there is the validation data
        if validation_data is not None:

            # Evaluate the model:
            self.evaluate(*validation_data,
                          batch_size=batch_size)

    # Evaluates the model using passed in dataset
    def evaluate(self, X_val, y_val, *, batch_size=None):

        # Default value if batch size is not being set
        validation_steps = 1

        # Calculate number of steps
        if batch_size is not None:
            validation_steps = len(X_val) // batch_size
            # Dividing rounds down. If there are some remaining
            # data but not a full batch, this won't include it
            # Add `1` to include this not full batch
            if validation_steps * batch_size < len(X_val):
                validation_steps += 1

        # Reset accumulated values in loss
        # and accuracy objects
        self.loss.new_pass()
        self.accuracy.new_pass()
```

```
        # Iterate over steps
        for step in range(validation_steps):

            # If batch size is not set -
            # train using one step and full dataset
            if batch_size is None:
                batch_X = X_val
                batch_y = y_val

            # Otherwise slice a batch
            else:
                batch_X = X_val[
                    step*batch_size:(step+1)*batch_size
                ]
                batch_y = y_val[
                    step*batch_size:(step+1)*batch_size
                ]

            # Perform the forward pass
            output = self.forward(batch_X, training=False)

            # Calculate the loss
            self.loss.calculate(output, batch_y)

            # Get predictions and calculate an accuracy
            predictions = self.output_layer_activation.predictions(
                              output)
            self.accuracy.calculate(predictions, batch_y)

        # Get and print validation loss and accuracy
        validation_loss = self.loss.calculate_accumulated()
        validation_accuracy = self.accuracy.calculate_accumulated()

        # Print a summary
        print(f'validation, ' +
              f'acc: {validation_accuracy:.3f}, ' +
              f'loss: {validation_loss:.3f}')

    # Predicts on the samples
    def predict(self, X, *, batch_size=None):

        # Default value if batch size is not being set
        prediction_steps = 1

        # Calculate number of steps
        if batch_size is not None:
            prediction_steps = len(X) // batch_size
```

```
            # Dividing rounds down. If there are some remaining
            # data but not a full batch, this won't include it
            # Add `1` to include this not full batch
            if prediction_steps * batch_size < len(X):
                prediction_steps += 1

        # Model outputs
        output = []

        # Iterate over steps
        for step in range(prediction_steps):

            # If batch size is not set -
            # train using one step and full dataset
            if batch_size is None:
                batch_X = X

            # Otherwise slice a batch
            else:
                batch_X = X[step*batch_size:(step+1)*batch_size]

            # Perform the forward pass
            batch_output = self.forward(batch_X, training=False)

            # Append batch prediction to the list of predictions
            output.append(batch_output)

        # Stack and return results
        return np.vstack(output)

    # Performs forward pass
    def forward(self, X, training):

        # Call forward method on the input layer
        # this will set the output property that
        # the first layer in "prev" object is expecting
        self.input_layer.forward(X, training)

        # Call forward method of every object in a chain
        # Pass output of the previous object as a parameter
        for layer in self.layers:
            layer.forward(layer.prev.output, training)

        # "layer" is now the last object from the list,
        # return its output
        return layer.output
```

```
    # Performs backward pass
    def backward(self, output, y):

        # If softmax classifier
        if self.softmax_classifier_output is not None:
            # First call backward method
            # on the combined activation/loss
            # this will set dinputs property
            self.softmax_classifier_output.backward(output, y)

            # Since we'll not call backward method of the last layer
            # which is Softmax activation
            # as we used combined activation/loss
            # object, let's set dinputs in this object
            self.layers[-1].dinputs = \
                self.softmax_classifier_output.dinputs

            # Call backward method going through
            # all the objects but last
            # in reversed order passing dinputs as a parameter
            for layer in reversed(self.layers[:-1]):
                layer.backward(layer.next.dinputs)

            return

        # First call backward method on the loss
        # this will set dinputs property that the last
        # layer will try to access shortly
        self.loss.backward(output, y)

        # Call backward method going through all the objects
        # in reversed order passing dinputs as a parameter
        for layer in reversed(self.layers):
            layer.backward(layer.next.dinputs)

    # Retrieves and returns parameters of trainable layers
    def get_parameters(self):

        # Create a list for parameters
        parameters = []

        # Iterable trainable layers and get their parameters
        for layer in self.trainable_layers:
            parameters.append(layer.get_parameters())

        # Return a list
        return parameters
```

```
    # Updates the model with new parameters
    def set_parameters(self, parameters):

        # Iterate over the parameters and layers
        # and update each layers with each set of the parameters
        for parameter_set, layer in zip(parameters,
                                        self.trainable_layers):
            layer.set_parameters(*parameter_set)

    # Saves the parameters to a file
    def save_parameters(self, path):

        # Open a file in the binary-write mode
        # and save parameters into it
        with open(path, 'wb') as f:
            pickle.dump(self.get_parameters(), f)

    # Loads the weights and updates a model instance with them
    def load_parameters(self, path):

        # Open file in the binary-read mode,
        # load weights and update trainable layers
        with open(path, 'rb') as f:
            self.set_parameters(pickle.load(f))

    # Saves the model
    def save(self, path):

        # Make a deep copy of current model instance
        model = copy.deepcopy(self)

        # Reset accumulated values in loss and accuracy objects
        model.loss.new_pass()
        model.accuracy.new_pass()

        # Remove data from the input layer
        # and gradients from the loss object
        model.input_layer.__dict__.pop('output', None)
        model.loss.__dict__.pop('dinputs', None)

        # For each layer remove inputs, output and dinputs properties
        for layer in model.layers:
            for property in ['inputs', 'output', 'dinputs',
                             'dweights', 'dbiases']:
                layer.__dict__.pop(property, None)

        # Open a file in the binary-write mode and save the model
        with open(path, 'wb') as f:
            pickle.dump(model, f)
```

```
    # Loads and returns a model
    @staticmethod
    def load(path):

        # Open file in the binary-read mode, load a model
        with open(path, 'rb') as f:
            model = pickle.load(f)

        # Return a model
        return model


# Loads a MNIST dataset
def load_mnist_dataset(dataset, path):

    # Scan all the directories and create a list of labels
    labels = os.listdir(os.path.join(path, dataset))

    # Create lists for samples and labels
    X = []
    y = []

    # For each label folder
    for label in labels:
        # And for each image in given folder
        for file in os.listdir(os.path.join(path, dataset, label)):
            # Read the image
            image = cv2.imread(
                        os.path.join(path, dataset, label, file),
                        cv2.IMREAD_UNCHANGED)

            # And append it and a label to the lists
            X.append(image)
            y.append(label)

    # Convert the data to proper numpy arrays and return
    return np.array(X), np.array(y).astype('uint8')


# MNIST dataset (train + test)
def create_data_mnist(path):

    # Load both sets separately
    X, y = load_mnist_dataset('train', path)
    X_test, y_test = load_mnist_dataset('test', path)

    # And return all the data
    return X, y, X_test, y_test
```

```
# Label index to label name relation
fashion_mnist_labels = {
    0: 'T-shirt/top',
    1: 'Trouser',
    2: 'Pullover',
    3: 'Dress',
    4: 'Coat',
    5: 'Sandal',
    6: 'Shirt',
    7: 'Sneaker',
    8: 'Bag',
    9: 'Ankle boot'
}

# Read an image
image_data = cv2.imread('pants.png', cv2.IMREAD_GRAYSCALE)

# Resize to the same size as Fashion MNIST images
image_data = cv2.resize(image_data, (28, 28))

# Invert image colors
image_data = 255 - image_data

# Reshape and scale pixel data
image_data = (image_data.reshape(1, -1).astype(np.float32) -
                127.5) / 127.5

# Load the model
model = Model.load('fashion_mnist.model')

# Predict on the image
confidences = model.predict(image_data)

# Get prediction instead of confidence levels
predictions = model.output_layer_activation.predictions(confidences)

# Get label name from label index
prediction = fashion_mnist_labels[predictions[0]]

print(prediction)
```