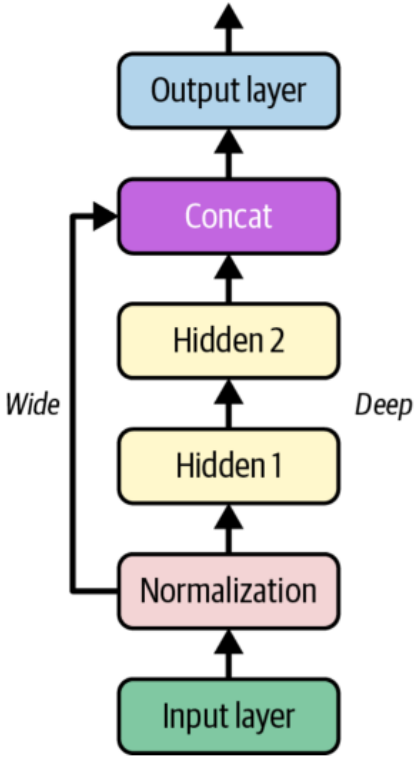
Complex Models using the Functional API

- The sequential API is clean and straightforward and sequential models are common.
- Sometimes we need neural networks with more complex topologies, or with multiple inputs or outputs.
  - For this purpose, Keras offers the functional API.
- One example of a non-sequential neural network is a *Wide & Deep* neural network.
  - This neural network architecture was introduced in a 2016 paper by Heng-Tze Cheng et al.
- It connects all or part of the inputs directly to the output layer.

Wide & Deep Neural Network

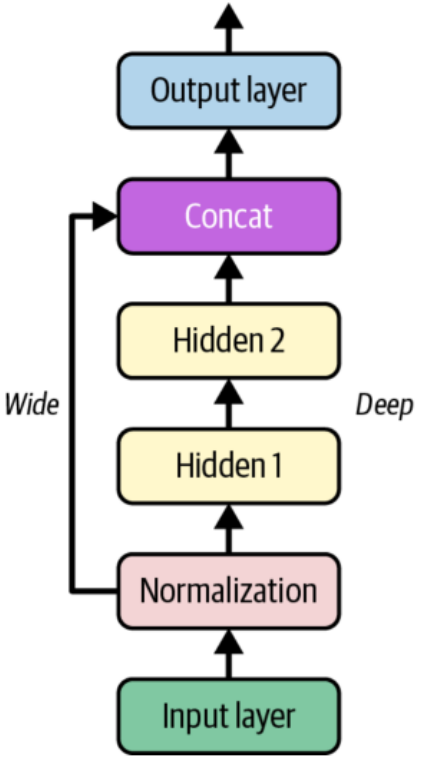
- This architecture makes it possible for the neural network to learn both deep patterns (using the deep path) and simple rules (through the short path).

- In contrast, a regular MLP forces all the data to flow through the full stack of layers.
  - Simple patterns in the data may end up being distorted by this sequence of transformations.



Wide & Deep Neural Network

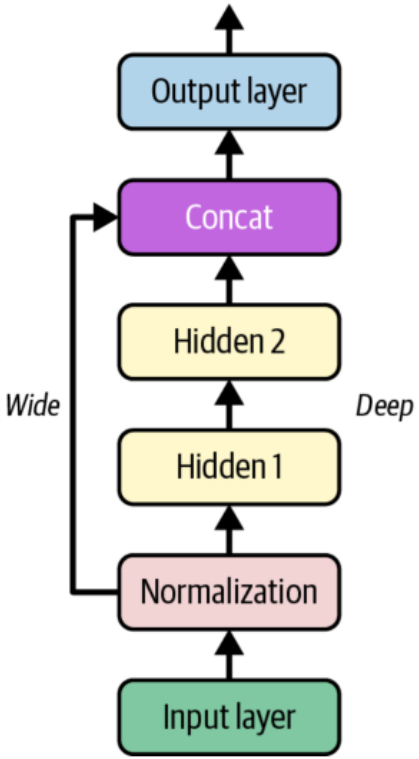
```
normalization_layer = tf.keras.layers.Normalization()
hidden_layer1 = tf.keras.layers.Dense(30, activation="relu")
hidden_layer2 = tf.keras.layers.Dense(30, activation="relu")
concat_layer = tf.keras.layers.Concatenate()
output_layer = tf.keras.layers.Dense(1)

input_ = tf.keras.layers.Input(shape=X_train.shape[1:])
normalized = normalization_layer(input_)
hidden1 = hidden_layer1(normalized)
hidden2 = hidden_layer2(hidden1)
concat = concat_layer([normalized, hidden2])
output = output_layer(concat)

model = tf.keras.Model(inputs=[input_], outputs=[output])
```



Model Summary

```
model.summary()
```

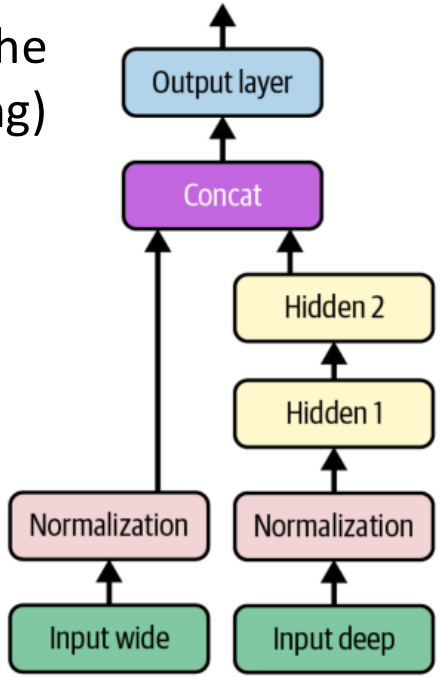
```
Model: "model"
_________________________________________________________________________________________________
 Layer (type)                   Output Shape         Param #     Connected to
==================================================================================================
 input_1 (InputLayer)           [(None, 8)]          0           []

 normalization (Normalization)  (None, 8)            17          ['input_1[0][0]']

 dense (Dense)                  (None, 30)           270         ['normalization[0][0]']

 dense_1 (Dense)                (None, 30)           930         ['dense[0][0]']

 concatenate (Concatenate)      (None, 38)           0           ['input_1[0][0]',
                                                                  'dense_1[0][0]']

 dense_2 (Dense)                (None, 1)            39          ['concatenate[0][0]']

==================================================================================================
Total params: 1,256
Trainable params: 1,239
Non-trainable params: 17
_________________________________________________________________________________________________
```



Handling Multiple Inputs

- We want to send a subset of the features through the wide path and a different subset (possibly overlapping) through the deep path:

```
input_wide = tf.keras.layers.Input(shape=[5])  # features 0 to 4
input_deep = tf.keras.layers.Input(shape=[6])  # features 2 to 7
norm_layer_wide = tf.keras.layers.Normalization()
norm_layer_deep = tf.keras.layers.Normalization()
norm_wide = norm_layer_wide(input_wide)
norm_deep = norm_layer_deep(input_deep)
hidden1 = tf.keras.layers.Dense(30, activation="relu")(norm_deep)
hidden2 = tf.keras.layers.Dense(30, activation="relu")(hidden1)
concat = tf.keras.layers.concatenate([norm_wide, hidden2])
output = tf.keras.layers.Dense(1)(concat)
model = tf.keras.Model(inputs=[input_wide, input_deep], outputs=[output])
```



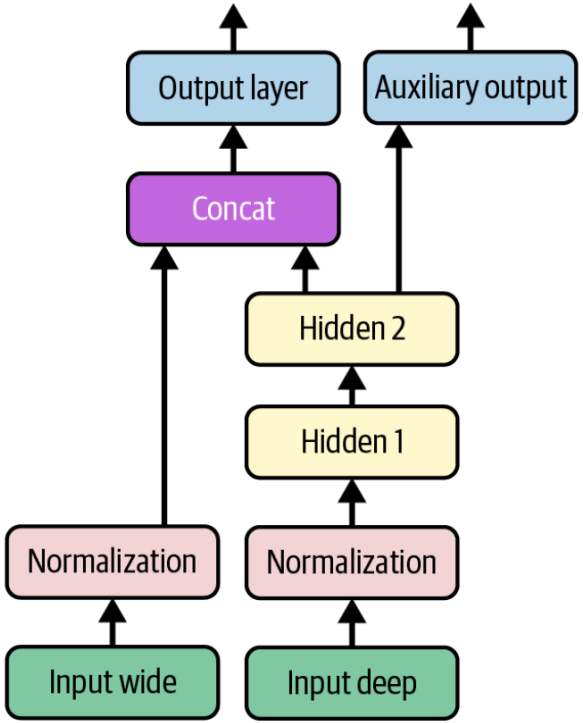
Handling Multiple Inputs

- When we call the `fit()` method, instead of passing a single input matrix `X_train`, we pass a pair of matrices (`X_train_wide, X_train_deep`).
- The same is true for `X_valid`, and also for `X_test` and `X_new` when you call `evaluate()` or `predict()`:

```
optimizer = tf.keras.optimizers.Adam(learning_rate=1e-3)
model.compile(loss="mse", optimizer=optimizer, metrics=["RootMeanSquaredError"])

X_train_wide, X_train_deep = X_train[:, :5], X_train[:, 2:]
X_valid_wide, X_valid_deep = X_valid[:, :5], X_valid[:, 2:]
X_test_wide, X_test_deep = X_test[:, :5], X_test[:, 2:]
X_new_wide, X_new_deep = X_test_wide[:3], X_test_deep[:3]

norm_layer_wide.adapt(X_train_wide)
norm_layer_deep.adapt(X_train_deep)
history = model.fit((X_train_wide, X_train_deep), y_train, epochs=20,
                    validation_data=((X_valid_wide, X_valid_deep), y_valid))
mse_test = model.evaluate((X_test_wide, X_test_deep), y_test)
y_pred = model.predict((X_new_wide, X_new_deep))
```

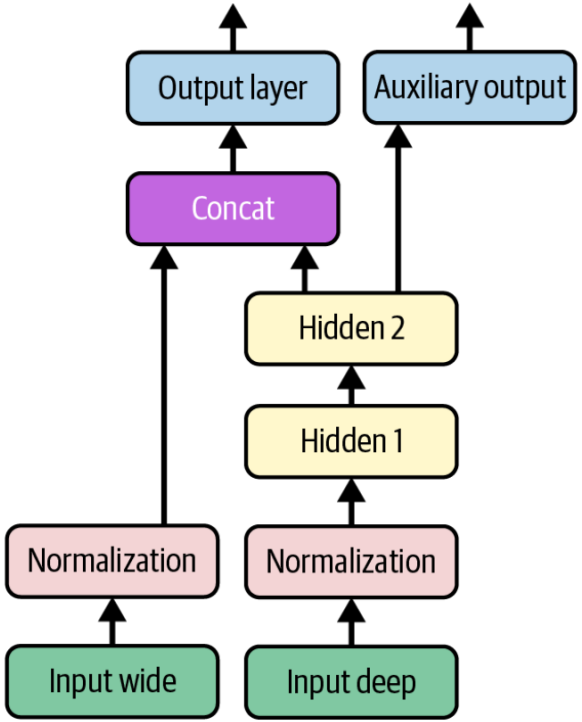
Having Multiple Outputs

- Cases in which you may have multiple outputs:
  - The task may demand it. e.g. you may want to locate and classify the main object in a picture.
  - You may have multiple independent tasks based on the same data.
  - As a regularization technique i.e., a training constraint whose objective is to reduce overfitting and thus improve the model's ability to generalize.



Having Multiple Outputs

```
input_wide = tf.keras.layers.Input(shape=[5])  # features 0 to 4
input_deep = tf.keras.layers.Input(shape=[6])  # features 2 to 7
norm_layer_wide = tf.keras.layers.Normalization()
norm_layer_deep = tf.keras.layers.Normalization()
norm_wide = norm_layer_wide(input_wide)
norm_deep = norm_layer_deep(input_deep)
hidden1 = tf.keras.layers.Dense(30, activation="relu")(norm_deep)
hidden2 = tf.keras.layers.Dense(30, activation="relu")(hidden1)
concat = tf.keras.layers.concatenate([norm_wide, hidden2])
output = tf.keras.layers.Dense(1)(concat)
aux_output = tf.keras.layers.Dense(1)(hidden2)
model = tf.keras.Model(inputs=[input_wide, input_deep],
                       outputs=[output, aux_output])
```



Handling Multiple Outputs

- Each output will need its own loss function. Therefore, when we compile the model, we should pass a list of losses.
  - If we pass a single loss, Keras will assume that the same loss for all outputs.
- By default, Keras will compute all the losses and simply add them up to get the final loss used for training.
- Since we care more about the main output than the auxiliary output, we want to give the main output's loss a much greater weight:

```
optimizer = tf.keras.optimizers.Adam(learning_rate=1e-3)
model.compile(loss=("mse", "mse"), loss_weights=(0.9, 0.1), optimizer=optimizer,
              metrics=["RootMeanSquaredError"])
```

Handling Multiple Outputs

- When we train the model, we need to provide labels for each output.
  - In this example, the main output and the auxiliary output should try to predict the same thing, so they should use the same labels.
- So instead of passing `y_train`, we need to pass (`y_train, y_train`):

```
norm_layer_wide.adapt(X_train_wide)
norm_layer_deep.adapt(X_train_deep)
history = model.fit(
    (X_train_wide, X_train_deep), (y_train, y_train), epochs=20,
    validation_data=((X_valid_wide, X_valid_deep), (y_valid, y_valid))
)
```

- When we evaluate the model, Keras returns the weighted sum of the losses, as well as all the individual losses and metrics:

```
eval_results = model.evaluate((X_test_wide, X_test_deep), (y_test, y_test))
weighted_sum_of_losses, main_loss, aux_loss, main_rmse, aux_rmse = eval_results
```

Handling Multiple Outputs

- If you set `return_dict=True`, then `evaluate`() will return a dictionary instead of a big tuple.
- Similarly, the `predict`() method will return predictions for each output:

```
y_pred_main, y_pred_aux = model.predict((X_new_wide, X_new_deep))
```

- The `predict`() method returns a tuple, and it does not have a `return_dict` argument to get a dictionary instead.
  - However, you can create one using `model.output_names`:

```
y_pred_tuple = model.predict((X_new_wide, X_new_deep))
y_pred = dict(zip(model.output_names, y_pred_tuple))
```

Declarative APIs

- The sequential API and the functional API are declarative.
  - You start by declaring which layers you want to use and how they should be connected, and then can you start feeding the model data for training.
- Advantages:
  - The model can easily be saved, cloned, and shared.
  - Its structure can be displayed and analyzed.
  - The framework can infer shapes and check types, so errors can be caught early (i.e., before any data ever goes through the model).
  - It's straightforward to debug, since the whole model is a static graph of layers.
- Disadvantage: it's static.
  - Some models involve loops, varying shapes, conditional branching, and other dynamic behaviors.

Subclassing API

- You subclass the `Model` class, create the layers in the constructor, and use them to perform the computations you want in the `call`() method.
- Creating an instance of the following `WideAndDeepModel` class gives us an equivalent model to the one we just built with the functional API:

```
class WideAndDeepModel(tf.keras.Model):
    def __init__(self, units=30, activation="relu", **kwargs):
        super().__init__(**kwargs)  # needed to support naming the model
        self.norm_layer_wide = tf.keras.layers.Normalization()
        self.norm_layer_deep = tf.keras.layers.Normalization()
        self.hidden1 = tf.keras.layers.Dense(units, activation=activation)
        self.hidden2 = tf.keras.layers.Dense(units, activation=activation)
        self.main_output = tf.keras.layers.Dense(1)
        self.aux_output = tf.keras.layers.Dense(1)
```

Subclassing API

- We separate the creation of the layers in the constructor from their usage in the `call`() method.
- We don’t need to create the input objects, we can use the input argument to the `call`() method.

```
    def call(self, inputs):
        input_wide, input_deep = inputs
        norm_wide = self.norm_layer_wide(input_wide)
        norm_deep = self.norm_layer_deep(input_deep)
        hidden1 = self.hidden1(norm_deep)
        hidden2 = self.hidden2(hidden1)
        concat = tf.keras.layers.concatenate([norm_wide, hidden2])
        output = self.main_output(concat)
        aux_output = self.aux_output(hidden2)
        return output, aux_output

model = WideAndDeepModel(30, activation="relu", name="my_cool_model")
```

Subclassing API

- After having a model instance, we can compile it, adapt its normalization layers (e.g., using `model.norm_layer_wide.adapt(...)` and `model.norm_layer_deep.adapt(...)`), fit it, evaluate it, and use it to make predictions, exactly like we did with the functional API.

- The big difference with subclassing API is that you can include anything you want in the `call()` method: for loops, if statements, low-level TensorFlow operations.

- This makes it a great API when experimenting with new ideas, especially for researchers.

Disadvantages of Subclassing API

- Your model's architecture is hidden within the `call`() method, so Keras cannot easily inspect it; the model cannot be cloned using `tf.keras.models.clone_model`().
- When you call the `summary`() method, you only get a list of layers, without any information on how they are connected to each other.
- Keras cannot check types and shapes ahead of time, and it is easier to make mistakes.
- So unless you really need that extra flexibility, you should probably stick to the sequential API or the functional API.

Saving and Restoring a Model

- Saving a trained Keras model is as simple as it gets:

```
model.save("my_keras_model", save_format="tf")
```

- When you set `save_format="tf"`, Keras saves the model using TensorFlow's *Saved-Model* format: this is a directory (with the given name) containing several files and subdirectories:
  - The *saved_model.pb* file contains the model's architecture and logic in the form of a serialized computation graph.
  - The *keras_metadata.pb* file contains extra information needed by Keras.
  - The *variables* subdirectory contains all the parameter values (including the connection weights, the biases, the normalization statistics, and the optimizer's hyperparameters).
  - The *assets* directory may contain extra files, such as data samples, feature names, class names, …

Saving and Restoring a Model

- Loading the model is just as easy as saving it:

```
model = tf.keras.models.load_model("my_keras_model")
y_pred_main, y_pred_aux = model.predict((X_new_wide, X_new_deep))
```

- You can also use `save_weights()` and `load_weights()` to save and load only the parameter values.
  - includes the connection weights, biases, preprocessing stats, optimizer state
- The parameter values are saved in one or more files such as *my_weights.data-00004-of-00052*, plus an index file *my_weights.index*.
- Saving just the weights is faster and uses less disk space than saving the whole model, so it's perfect to save quick checkpoints during training.
- If you're training a big model, and it takes hours or days, then you must save checkpoints regularly in case the computer crashes.

Callbacks

- The `fit()` method accepts a `callbacks` argument that lets you specify a list of objects that Keras will call before and after training, before and after each epoch, and even before and after processing each batch.
- The `ModelCheckpoint` callback saves checkpoints of your model at regular intervals during training, by default at the end of each epoch:

```
checkpoint_cb = tf.keras.callbacks.ModelCheckpoint("my_checkpoints",
                                                   save_weights_only=True)
history = model.fit([...], callbacks=[checkpoint_cb])
```

- If you use a validation set during training, you can set `save_best_only=True` when creating the `ModelCheckpoint`.
  - It will only save your model when its performance on the validation set is the best so far.

`EarlyStopping` Callback

- `EarlyStopping` callback will interrupt training when it measures no progress on the validation set for a number of epochs (defined by the patience argument).
  - If you set `restore_best_weights=True` it will roll back to the best model at the end of training.

- You can combine both callbacks to save checkpoints of your model, and interrupt training early when there is no more progress, to avoid wasting time and resources and to reduce overfitting:

```
early_stopping_cb = tf.keras.callbacks.EarlyStopping(patience=10,
                                                     restore_best_weights=True)
history = model.fit([...], callbacks=[checkpoint_cb, early_stopping_cb])
```

4. Fine-Tuning Neural Network Hyperparameters

The Curse of Flexibility

- The flexibility of neural networks is also one of their main drawbacks: there are many hyperparameters to tweak.
- Not only can you use any imaginable network architecture, but even in a basic MLP you can change:
  - the number of layers
  - the number of neurons
  - the type of activation function to use in each layer
  - the weight initialization logic
  - the type of optimizer to use
  - the learning rate
  - the batch size
- How do you know what combination of hyperparameters is the best for your task?

Hyperparameter Tuning

- One option is to convert your Keras model to a Scikit-Learn estimator, and then use `GridSearchCV` or `RandomizedSearchCV`.
  - For this, you can use the `KerasRegressor` and `KerasClassifier` wrapper classes from the *SciKeras* library.
- The better way: you can use the *Keras Tuner* library, which is a hyperparameter tuning library for Keras models.
  - It offers several tuning strategies, and is highly customizable.
  - Import `keras_tuner`, usually as `kt`, then write a function that builds, compiles, and returns a Keras model.
  - The function must take a `kt.HyperParameters` object as an argument, which it can use to define hyperparameters along with their range of possible values.

Using Keras Tuner

```
import keras_tuner as kt

def build_model(hp):
    n_hidden = hp.Int("n_hidden", min_value=0, max_value=8, default=2)
    n_neurons = hp.Int("n_neurons", min_value=16, max_value=256)
    learning_rate = hp.Float("learning_rate", min_value=1e-4, max_value=1e-2,
                             sampling="log")
    optimizer = hp.Choice("optimizer", values=["sgd", "adam"])
    if optimizer == "sgd":
        optimizer = tf.keras.optimizers.SGD(learning_rate=learning_rate)
    else:
        optimizer = tf.keras.optimizers.Adam(learning_rate=learning_rate)

    model = tf.keras.Sequential()
    model.add(tf.keras.layers.Flatten())
    for _ in range(n_hidden):
        model.add(tf.keras.layers.Dense(n_neurons, activation="relu"))
    model.add(tf.keras.layers.Dense(10, activation="softmax"))
    model.compile(loss="sparse_categorical_crossentropy", optimizer=optimizer,
                  metrics=["accuracy"])
    return model
```

Using Keras Tuner

- To do a basic random search, you can create a `kt.RandomSearch` tuner, passing the `build_model` function to the constructor, and call the tuner's `search`() method:

```
random_search_tuner = kt.RandomSearch(
    build_model, objective="val_accuracy", max_trials=5, overwrite=True,
    directory="my_fashion_mnist", project_name="my_rnd_search", seed=42)
random_search_tuner.search(X_train, y_train, epochs=10,
                           validation_data=(X_valid, y_valid))
```

- Since `objective` is set to "`val_accuracy`", the tuner prefers models with a higher validation accuracy, so once the tuner has finished searching, you can get the best models:

```
top3_models = random_search_tuner.get_best_models(num_models=3)
best_model = top3_models[0]
```

Using Keras Tuner

- You can also call `get_best_hyperparameters()` to get the `kt.HyperParameters` of the best models:

```
top3_params = random_search_tuner.get_best_hyperparameters(num_trials=3)
top3_params[0].values  # best hyperparameter values
```

```
{'n_hidden': 5,
 'n_neurons': 70,
 'learning_rate': 0.00041268008323824807,
 'optimizer': 'adam'}
```

- Each tuner is guided by a so-called *oracle*: before each trial, the tuner asks the oracle to tell it what the next trial should be.
- The `RandomSearch` tuner uses a `RandomSearchOracle`, which is pretty basic: it just picks the next trial randomly.

Using Keras Tuner

- Since the oracle keeps track of all the trials, you can ask it to give you the best one, and you can display a summary of that trial:

```
best_trial = random_search_tuner.oracle.get_best_trials(num_trials=1)[0]
best_trial.summary()
```

```
Trial summary
Hyperparameters:
n_hidden: 5
n_neurons: 70
learning_rate: 0.00041268008323824807
optimizer: adam
Score: 0.8736000061035156
```

- You can access the metrics directly:

```
best_trial.metrics.get_last_value("val_accuracy")
```

- If you are happy with the best model's performance, continue training it for a few epochs on the full training set, then evaluate it on the test set:

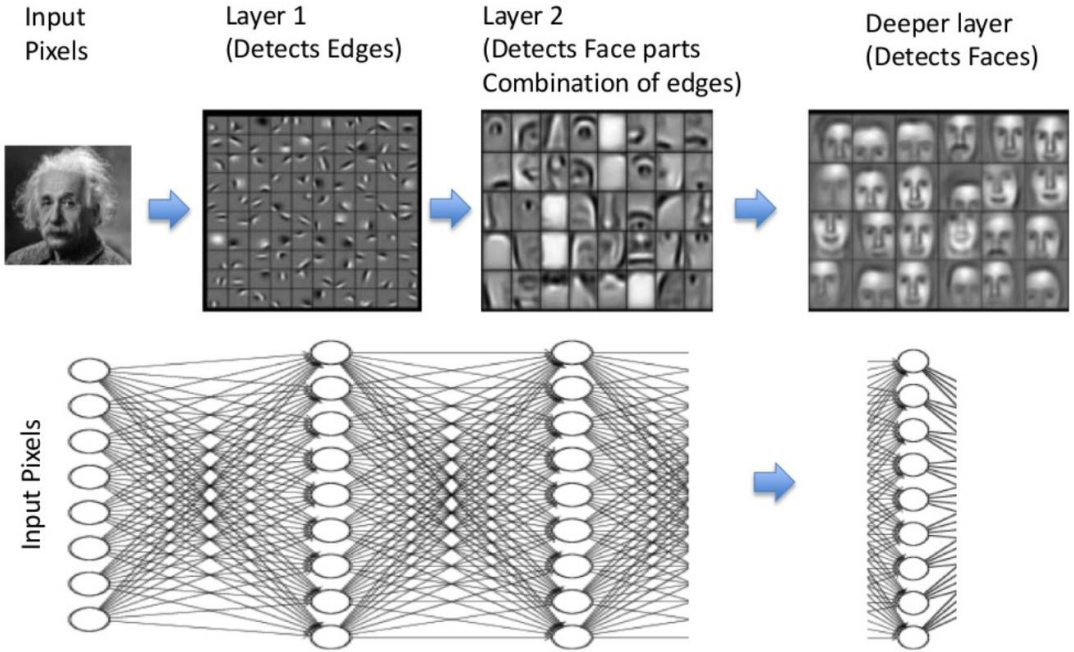
```
best_model.fit(X_train_full, y_train_full, epochs=10)
test_loss, test_accuracy = best_model.evaluate(X_test, y_test)
```

Number of Hidden Layers

- For many problems, you can begin with a single hidden layer and get reasonable results.
- An MLP with just one hidden layer can theoretically model even the most complex functions, provided it has enough neurons.
- For complex problems, deep networks have a much higher *parameter efficiency* than shallow ones:
  - They can model complex functions using exponentially fewer neurons than shallow nets, allowing them to reach much better performance with the same amount of training data.

Number of Hidden Layers

- Real-world data is often structured in a hierarchical way, and deep neural networks take advantage of this fact:



Number of Hidden Layers

- This hierarchical architecture helps DNNs converge faster to a good solution, and improves their ability to generalize to new datasets.
  - e.g., if you have already trained a model to recognize faces in pictures and you now want to train a new neural network to recognize hairstyles, you can start the training by reusing the lower layers of the first network.
- Instead of randomly initializing the weights and biases of the first few layers of the new neural network, you can initialize them to the values of the weights and biases of the lower layers of the first network.
  - This way the network will not have to learn from scratch all the low-level structures that occur in most pictures; it will only have to learn the higher-level structures (e.g., hairstyles).
- This is called *transfer learning*.

Number of Hidden Layers

- For many problems you can start with just one or two hidden layers and the neural network will work just fine.
- For more complex problems, you can ramp up the number of hidden layers until you start overfitting the training set.
- Very complex tasks, such as large image classification, typically require networks with tens of layers (or even hundreds, but not fully connected ones) and they need a huge amount of training data.
- You will rarely have to train such networks from scratch: it is common to reuse parts of a pre-trained state-of-the-art network that performs a similar task.
  - Training will then be a lot faster and require much less data.

Number of Neurons per Hidden Layer

- The number of neurons in the input and output layers is determined by the type of input and output your task requires.
  - e.g. the MNIST task requires 28 × 28 = 784 inputs and 10 output neurons.
- For the hidden layers, it *used to be* common to size them to form a pyramid, with fewer and fewer neurons at each layer.
  - Rationale: low-level features can combine into fewer high-level features.
  - A typical neural network for MNIST might have 3 hidden layers, the first with 300 neurons, the second with 200, and the third with 100.
- This practice has been largely abandoned because:
  - It seems that using the same number of neurons in all hidden layers performs just as well in most cases, or even better.
  - There is only one hyperparameter to tune, instead of one per layer.

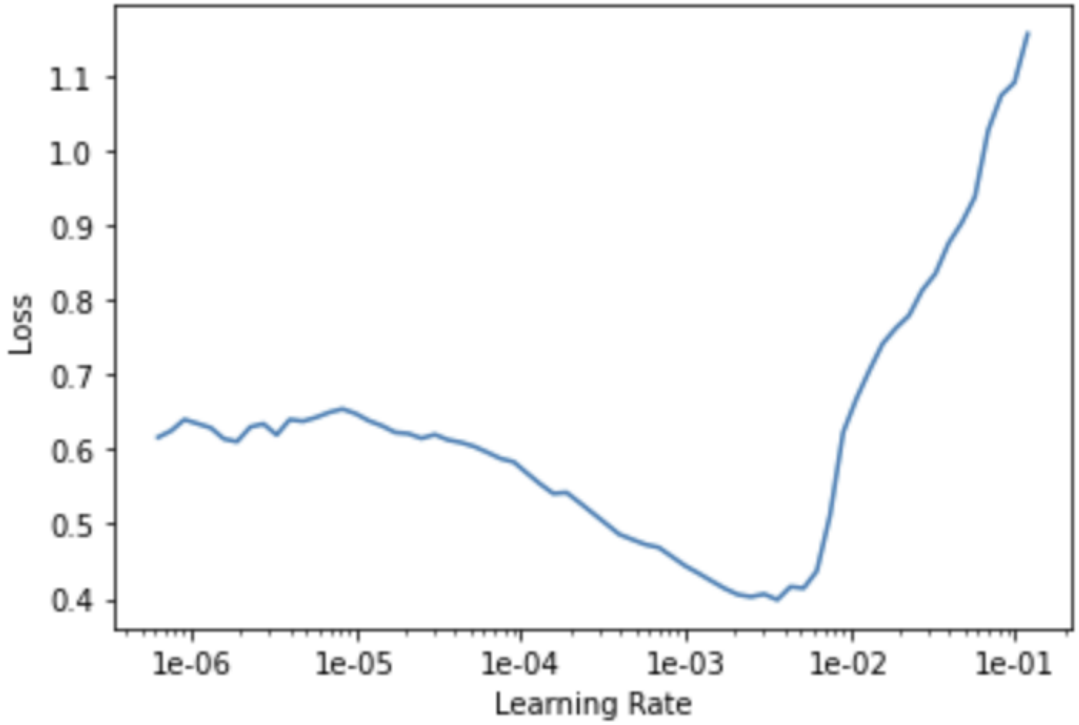
Number of Neurons per Hidden Layer

- You can try increasing the number of neurons gradually until the network starts overfitting.
  - Alternatively, you can try building a model with slightly more layers and neurons than you actually need, then use early-stopping and other regularization techniques to prevent it from overfitting.
- If a layer has too few neurons, it won't have enough representational power to preserve all the useful information from the inputs.
  - e.g., a layer with two neurons can only output 2D data, so if it gets 3D data as input, some information will be lost
  - No matter how big and powerful the rest of the network is, that information will never be recovered.

Learning Rate

- The learning rate is probably the most important hyperparameter.
- One way to find a good learning rate is to train the model for a few hundred iterations, starting with a very low learning rate (e.g., 10^{-5}) and gradually increasing it up to a very large value (e.g., 10).
- If you plot the loss as a function of the learning rate, you should see it dropping at first. But after a while, the learning rate will be too large, so the loss will shoot back up.
  - The optimal learning rate will be a bit lower than the point at which the loss starts to climb (typically about 10 times lower than the turning point).

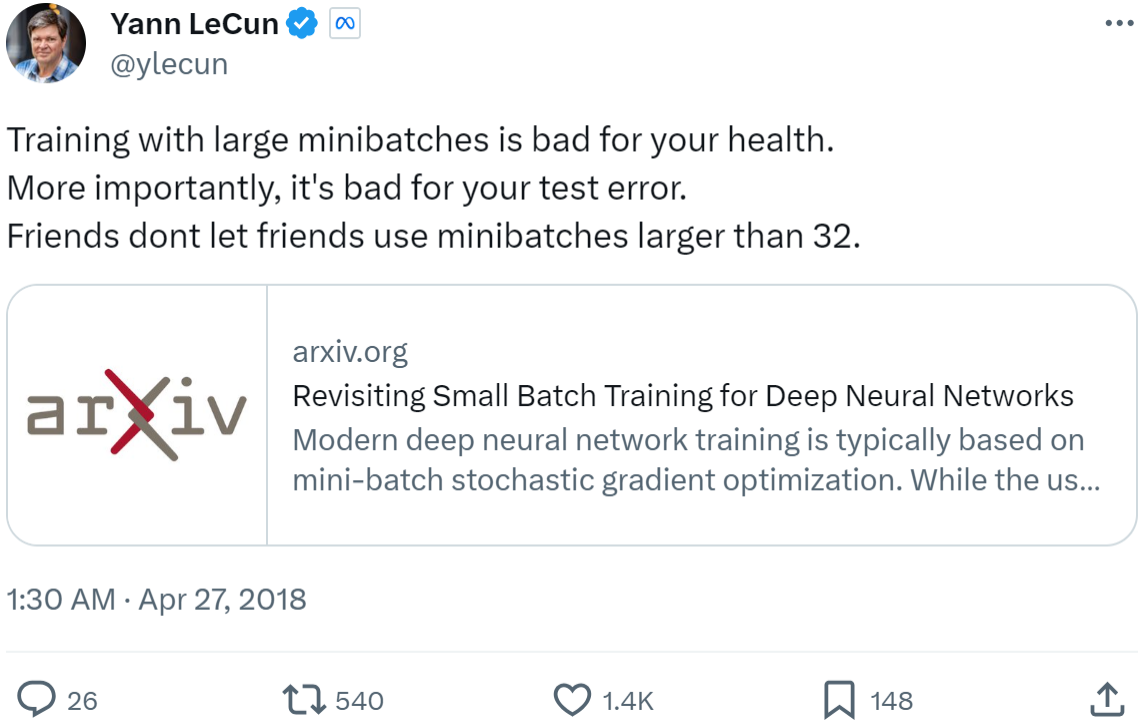
Loss vs. Learning Rate



Batch Size

- The batch size can have a significant impact on your model's performance and training time.
- Hardware accelerators can process large batch sizes efficiently, so the training algorithm will see more instances per second.
  - Many researchers and practitioners recommend using the largest batch size that can fit in GPU RAM.
- Problem: large batch sizes often lead to training instabilities, especially at the beginning of training, and the resulting model may not generalize well.
- Solution: use a large batch size, with learning rate warmup, and if training is unstable or the final performance is disappointing, then try using a small batch size instead.

Batch Size


Yann LeCun
@ylecun
Training with large minibatches is bad for your health.
More importantly, it's bad for your test error.
Friends dont let friends use minibatches larger than 32.
arxiv.org
Revisiting Small Batch Training for Deep Neural Networks
Modern deep neural network training is typically based on mini-batch stochastic gradient optimization. While the us...
1:30 AM · Apr 27, 2018
26
540
1.4K
148

Other Hyperparameters

- *Optimizer:* choosing a better optimizer than plain old mini-batch gradient descent (and tuning its hyperparameters) is important.
- *Activation function:* in general, the ReLU activation function will be a good default for all hidden layers, but for the output layer it really depends on your task.
- *Number of iterations:* in most cases, the number of training iterations does not actually need to be tweaked: just use early stopping instead.
- The optimal learning rate depends on the other hyperparameters, especially the batch size.
  - If you modify any hyperparameter, make sure to update the learning rate as well.