THE UNIVERSITY OF CHICAGO

Machine Learning and Predictive Analytics, MSCA-31009-4

Lecture 5

Igor Yakushin

April 30, 2021

Overview I

Overview II

- Number of Hidden Layers
- Number of Neurons per Hidden Layer
- Learning Rate, Batch Size, and Other Hyperparameters

Introduction to Artificial Neural Networks with Keras

Introduction to Artificial Neural Networks with Keras

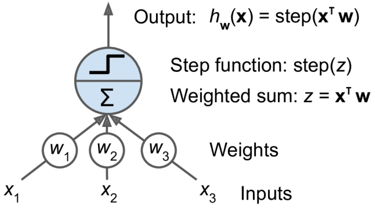
- Artificial Neural Networks (ANN) is a Machine Learning model inspired by the networks of biological neurons found in our brains
- However, ANNs have gradually become quite different from their biological cousins and it is easier now to consider them on their own
- ANNs were first introduced in 1943 by the neurophysiologist Warren McCulloch and the mathematician Walter Pitts
- They presented a simplified computational model of how biological neurons might work together in animal brains to perform complex computations using propositional logic.
- At first there was a lot of excitement and high expectations but by 1960s it became obvious that that approach was very limited and could not solve some simplest problems so the interest in ANNs disappeared.

Introduction to Artificial Neural Networks with Keras

- In 1980s new architectures, new ways to train the networks appeared that sparked a revival of interest in ANNs but the progress was slow, the computing power was not there yet and new powerful techniques like Support Vector Machines were invented that had better mathematical foundation and offerred better results at that time.
- We are now in the middle of the new wave of interest to ANNs that started in about 2012 but this time it is unlikely to go away any time soon
  - There is now a huge quantity of data available to train neural networks, and ANNs frequently outperform other ML techniques on very large and complex problems.
  - The tremendous increase in computing power since the 1990s now makes it possible to train large neural networks in a reasonable amount of time.
  - The training algorithms have been improved
  - ANNs seem to have entered a virtuous circle of funding and progress. Amazing products based on ANNs regularly make the headline news, which pulls more and more attention and funding toward them

The Perceptron

- The Perceptron is one of the simplest ANN architectures invented in 1957 by Frank Rosenblatt.
- It is based on a threshold logic unit (TLU), also called a linear threshold unit LTU
- TLU computes a weighted sum of its inputs, then applies a step function:



Output: $h_{\mathbf{w}}(\mathbf{x}) = \text{step}(\mathbf{x}^T \mathbf{w})$
Step function: step(z)
Weighted sum: $z = \mathbf{x}^T \mathbf{w}$
Σ
w_1
w_2
w_3
Weights
x_1
x_2
x_3
Inputs

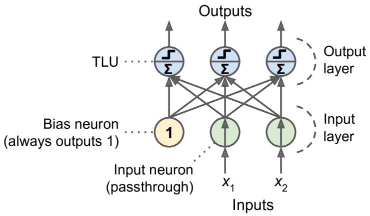
$$heaviside(z) = \begin{cases} 0, & \text{if } z < 0 \\ 1, & \text{if } z \geq 0 \end{cases} \tag{1}$$

$$sgn(z) = \begin{cases} -1, & \text{if } z < 0 \\ 0, & \text{if } z = 0 \\ 1, & \text{if } z > 0 \end{cases} \tag{2}$$

The Perceptron

- A single TLU can be used for simple linear binary classification. It computes a linear combination of the inputs, and if the result exceeds a threshold, it outputs the positive class. Otherwise it outputs the negative class.
- A Perceptron is simply composed of a single layer of TLUs, with each TLU connected to all the inputs. When all the neurons in a layer are connected to every neuron in the previous layer (i.e., its input neurons), the layer is called a fully connected layer, or a dense layer.
- The inputs of the Perceptron are fed to special passthrough neurons called input neurons: they output whatever input they are fed.
- All the input neurons form the input layer. Moreover, an extra bias feature is generally added ($x_0 = 1$): it is typically represented using a special type of neuron called a bias neuron, which outputs 1 all the time.

The Perceptron



One can compute all the outputs for all the instances in one formula:

$$h_{\mathbf{W},\mathbf{b}}(\mathbf{X}) = \phi(\mathbf{X}\mathbf{W} + \mathbf{b}) \qquad (3)$$

- **X** - matrix of input features, one row per instance, one column per feature
- **W** - contains all the connection weights except for the one from the bias neuron, one row per input neuron, one column per unit in the layer
- **b** contains all the connection weights between the bias neuron and the neurons in the output layer. It has one bias term per neuron in the output layer.
- ϕ - the activation function: when the artificial neurons are TLUs, it is a step function

The Perceptron learning rule

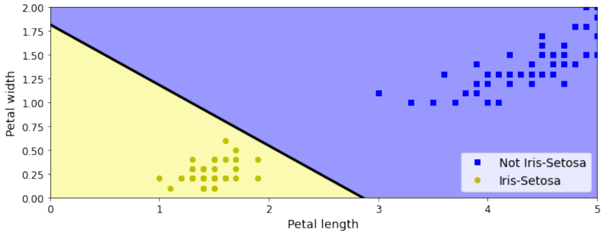
- The Perceptron is fed one training instance at a time, and for each instance it makes its predictions and adjusts the weights:

$$w_{i,j}^{\text{next step}} = w_{i,j} + \eta(y_j - \hat{y}_j)x_i \tag{4}$$

- $w_{i,j}$ - connection weight between ith input neuron and jth output neuron
- x_i - the ith input value for the current training instance
- $\hat{y}_j$ - output of the jth output neuron for the current training instance
- y_j - target output of the jth output neuron for the current training instance
- η - learning rate

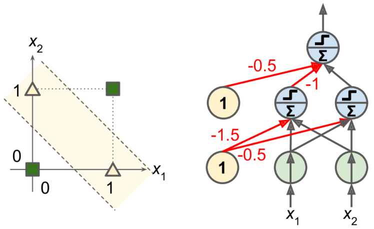
The Perceptron learning rule

- The decision boundary of each output neuron is linear, so Perceptrons are incapable of learning complex patterns
- However, if the training instances are linearly separable, this algorithm would converge to a solution
- Cells 2-4, l5_n1.ipynb



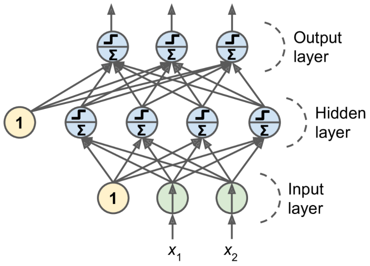
The Perceptron learning rule

- Note that contrary to Logistic Regression classifiers, Perceptrons do not output a class probability; rather, they make predictions based on a hard threshold. This is one reason to prefer Logistic Regression over Perceptrons.
- Perceptron, like any linear classifier, cannot find the correct solution if the classes are not linearly separable.
- However, stacking multiple Perceptrons into a Multilayer Perceptron (MLP) allows to handle complicated decision boundaries.



All connections have a weight equal to 1, except the four connections where the weight is shown.

The Multilayer Perceptron and Backpropagation



An MLP is composed of one (passthrough) input layer, one or more layers of TLUs, called hidden layers, and one final layer of TLUs called the output layer

- The layers close to the input layer are usually called the lower layers, and the ones close to the outputs are usually called the upper layers.
- Every layer except the output layer includes a bias neuron and is fully connected to the next layer.
- The signal flows only in one direction (from the inputs to the outputs), so this architecture is an example of a feedforward neural network (FNN).

The Multilayer Perceptron and Backpropagation

- When an ANN contains a deep stack of hidden layers, it is called a deep neural network (DNN).
- The field of Deep Learning studies DNNs, and more generally models containing deep stacks of computations.
- For many years researchers struggled to find a way to train MLPs, without success. But in 1986, David Rumelhart, Geoffrey Hinton, and Ronald Williams published a groundbreaking paper that introduced the backpropagation training algorithm, which is still used today
- Backpropagation is Gradient Descent using an efficient technique for computing the gradients: in just two passes through the network (one forward, one backward), the backpropagation algorithm is able to compute the gradient of the network's error with regard to every single model parameter.
- Once it has these gradients, it just performs a regular Gradient Descent step, and the whole process is repeated until the network converges to the solution.

The Multilayer Perceptron and Backpropagation

More details about backpropagation:

1. It handles one mini-batch at a time, and it goes through the full training set multiple times. Each pass is called an epoch.
2. Each mini-batch is passed to the network's input layer, which sends it to the first hidden layer. The algorithm then computes the output of all the neurons in this layer (for every instance in the mini-batch). The result is passed on to the next layer, its output is computed and passed to the next layer, and so on until we get the output of the last layer, the output layer. This is the forward pass: it is exactly like making predictions, except all intermediate results are preserved since they are needed for the backward pass.
3. Next, the algorithm measures the network's output error (i.e., it uses a loss function that compares the desired output and the actual output of the network, and returns some measure of the error).

The Multilayer Perceptron and Backpropagation

5. Then it computes how much each output connection contributed to the error. This is done analytically by applying the chain rule which makes this step fast and precise.
6. The algorithm then measures how much of these error contributions came from each connection in the layer below, again using the chain rule, working backward until the algorithm reaches the input layer. As explained earlier, this reverse pass efficiently measures the error gradient across all the connection weights in the network by propagating the error gradient backward through the network (hence the name of the algorithm).
7. Finally, the algorithm performs a Gradient Descent step to tweak all the connection weights in the network, using the error gradients it just computed.

The Multilayer Perceptron and Backpropagation

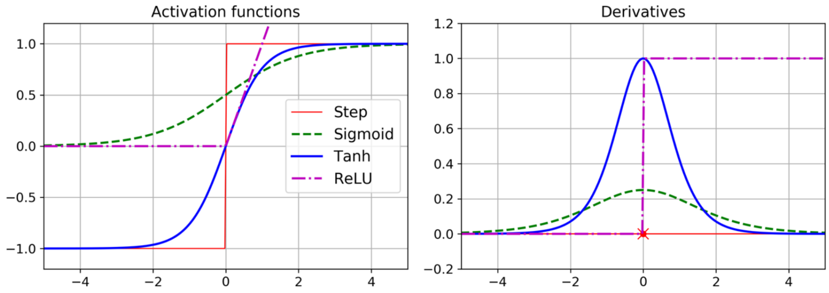
- It is important to initialize all the hidden layers' connection weights randomly, or else training will fail.
- For example, if you initialize all weights and biases to zero, then all neurons in a given layer will be perfectly identical, and thus backpropagation will affect them in exactly the same way, so they will remain identical.
- The step activation function in MLP is replaced with sigmoid: $\sigma(x) = \frac{1}{1+e^{-x}}$
- This was essential because the step function contains only flat segments, so there is no gradient to work with (Gradient Descent cannot move on a flat surface), while the logistic function has a well-defined nonzero derivative everywhere, allowing Gradient Descent to make some progress at every step.

Activation functions

Besides logistic function, other activation functions used in ANNs:

- The hyperbolic tangent function: $tanh(z) = 2\sigma(2z)–1$
  newline Just like the logistic function, this activation function is S-shaped, continuous, and differentiable, but its output value ranges from -1 to 1 (instead of 0 to 1 in the case of the logistic function). That range tends to make each layer's output more or less centered around 0 at the beginning of training, which often helps speed up convergence.
- The Rectified Linear Unit function: $ReLU(z) = max(0, z)$
  The ReLU function is continuous but unfortunately not differentiable at $z = 0$ (the slope changes abruptly, which can make Gradient Descent bounce around), and its derivative is 0 for $z < 0$. In practice, however, it works very well and has the advantage of being fast to compute, so it has become the default. Most importantly, the fact that it does not have a maximum output value helps reduce some issues during Gradient Descent

Activation functions



Regression MLPs

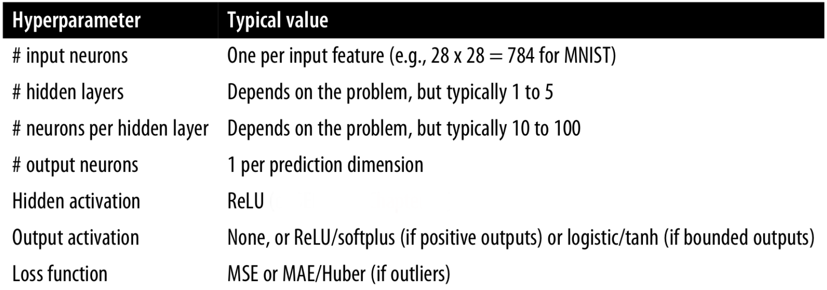
- MLPs can be used for regression tasks.
- If you want to predict a single value - for example, the price of a house, given many of its features - then you just need a single output neuron: its output is the predicted value.
- For multivariate regression - to predict multiple values at once - you need one output neuron per output dimension.
- For example, to locate the center of an object in an image, you need to predict 2D coordinates, so you need two output neurons.
- If you also want to place a bounding box around the object, then you need two more numbers: the width and the height of the object. So, you end up with four output neurons.

Regression MLPs

- In general, when building an MLP for regression, you do not want to use any activation function for the output neurons, so they are free to output any range of values.
- If you want to guarantee that the output will always be positive, then you can use the ReLU activation function in the output layer.
- Alternatively, you can use the softplus activation function, which is a smooth variant of ReLU: $softplus(z) = log(1 + exp(z))$. It is close to 0 when z is negative, and close to z when z is positive.
- Finally, if you want to guarantee that the predictions will fall within a given range of values, then you can use the logistic function or the hyperbolic tangent, and then scale the labels to the appropriate range: 0 to 1 for the logistic function and –1 to 1 for the hyperbolic tangent.
- The loss function to use during training is typically the mean squared error, but if you have a lot of outliers in the training set, you may prefer to use the mean absolute error instead. Alternatively, you can use the Huber loss, which is a combination of both.

Regression MLPs

- The Huber loss is quadratic when the error is smaller than a threshold δ (typically 1) but linear when the error is larger than δ. The linear part makes it less sensitive to outliers than the mean squared error, and the quadratic part allows it to converge faster and be more precise than the mean absolute error.



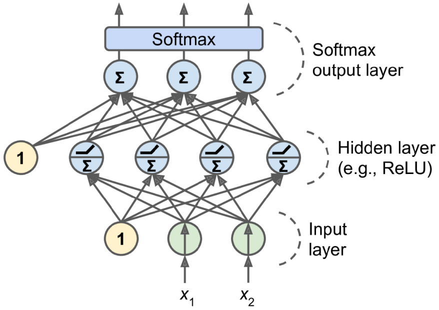
Hyperparameter	Typical value
# input neurons	One per input feature (e.g., 28 x 28 = 784 for MNIST)
# hidden layers	Depends on the problem, but typically 1 to 5
# neurons per hidden layer	Depends on the problem, but typically 10 to 100
# output neurons	1 per prediction dimension
Hidden activation	ReLU
Output activation	None, or ReLU/softplus (if positive outputs) or logistic/tanh (if bounded outputs)
Loss function	MSE or MAE/Huber (if outliers)

Classification MLPs

- For a binary classification problem, you just need a single output neuron using the logistic activation function: the output will be a number between 0 and 1, which you can interpret as the estimated probability of the positive class.
- MLPs can also easily handle multilabel binary classification tasks. For example, you could have an email classification system that predicts whether each incoming email is ham or spam, and simultaneously predicts whether it is an urgent or nonurgent email. In this case, you would need two output neurons, both using the logistic activation function: the first would output the probability that the email is spam, and the second would output the probability that it is urgent.
- Note that the output probabilities do not necessarily add up to 1.
- This lets the model output any combination of labels: you can have nonurgent ham, urgent ham, nonurgent spam, and perhaps even urgent spam.

Classification MLPs

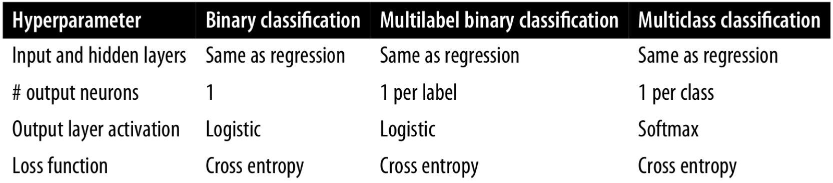
- If each instance can belong only to a single class, out of three or more possible classes (e.g., classes 0 through 9 for digit image classification), then you need to have one output neuron per class, and you should use the softmax activation function for the whole output layer.



- The softmax function will ensure that all the estimated probabilities are between 0 and 1 and that they add up to 1 (which is required if the classes are exclusive).
- This is called multiclass classification.

Classification MLPs

- Regarding the loss function, since we are predicting probability distributions, the cross-entropy loss (also called the log loss) is generally a good choice.



Hyperparameter	Binary classification	Multilabel binary classification	Multiclass classification
Input and hidden layers	Same as regression	Same as regression	Same as regression
# output neurons	1	1 per label	1 per class
Output layer activation	Logistic	Logistic	Softmax
Loss function	Cross entropy	Cross entropy	Cross entropy

- Play with various architectures and parameters here: `https://playground.tensorflow.org`

Implementing MLPs with Keras

- Keras is a high-level Deep Learning API that allows you to easily build, train, evaluate, and execute all sorts of neural networks.
- Documentation: `https://keras.io`
- The reference implementation, also called Keras, was developed by François Chollet and released in 2015
- The reference implementation could use as a backend lower level frameworks like TensorFlow, Theano, CNTK
- TensorFlow itself (developed by Google) now comes bundled with its own Keras implementation and that's what we will use
- It only supports TensorFlow as the backend, but it has the advantage of offering some very useful extra features : for example, it supports TensorFlow's Data API, which makes it easy to load and preprocess data efficiently.
- The most popular Deep Learning library, after Keras and TensorFlow, is Facebook's PyTorch library.

Building an Image Classifier Using the Sequential API

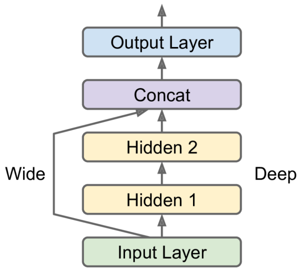
- Cells 9-45, l5_n1.ipynb

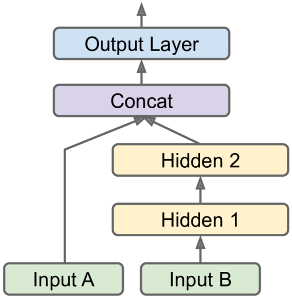
Building a Regression MLP Using the Sequential API

- Cells 48-52, l5_n1.ipynb

Building Complex Models Using the Functional API

- Cells 53-64, l5_n1.ipynb
- Wide & Deep NN: 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; thus, simple patterns in the data may end up being distorted by this sequence of transformations.





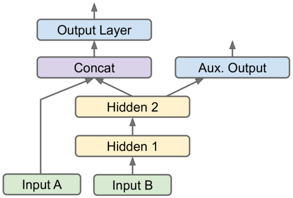
Building Complex Models Using the Functional API

There are many use cases in which you may want to have multiple outputs:

- For instance, you may want to locate and classify the main object in a picture. This is both a regression task (finding the coordinates of the object's center, as well as its width and height) and a classification task.
- Similarly, you may have multiple independent tasks based on the same data. Sure, you could train one neural network per task, but in many cases you will get better results on all tasks by training a single neural network with one output per task. This is because the neural network can learn features in the data that are useful across tasks. For example, you could perform multitask classification on pictures of faces, using one output to classify the person's facial expression (smiling, surprised, etc.) and another output to identify whether they are wearing glasses or not.

Building Complex Models Using the Functional API

- Another use case is as a regularization technique. For example, you may want to add some auxiliary outputs in a neural network architecture to ensure that the underlying part of the network learns something useful on its own, without relying on the rest of the network.



Using the Subclassing API to Build Dynamic Models

- Cells 65-66, l5_n1.ipynb
- Both 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 only then can you start feeding the model some data for training or inference.
- This has many 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 also fairly easy to debug, since the whole model is a static graph of layers.
- But the flip side is just that: it's static.
- Some models involve loops, varying shapes, conditional branching, and other dynamic behaviors. For such cases use the Subclassing API.

Using the Subclassing API to Build Dynamic Models

- This extra flexibility does come at a cost: your model's architecture is hidden within the call() method, so Keras cannot easily inspect it; it cannot save or clone it; and when you call the summary() method, you only get a list of layers, without any information on how they are connected to each other. Moreover, Keras cannot check types and shapes ahead of time, and it is easier to make mistakes.

Saving and Restoring a Model

- Cells 67-74, l5_n1.ipynb
- When using the Sequential API or the Functional API, saving a trained Keras model is simple: model.save("mymodel.h5")
- Keras will use the HDF5 format to save both the model's architecture (including every layer's hyperparameters) and the values of all the model parameters for every layer (e.g., connection weights and biases). It also saves the optimizer (including its hyperparameters and any state it may have)
- To load the model from file:
  model = keras.models.load_model("mymodel.h5")
- This will work when using the Sequential API or the Functional API, but unfortunately not when using model subclassing. You can use save_weights() and load_weights() to at least save and restore the model parameters, but you will need to save and restore everything else yourself.

Using Callbacks

- But what if training lasts several hours? This is quite common, especially when training on large datasets. In this case, you should not only save your model at the end of training, but also save checkpoints at regular intervals during training, to avoid losing everything if your computer crashes. But how can you tell the fit() method to save checkpoints? Use callbacks.
- Cells 75-80, l5_n1.ipynb
- The fit() method accepts a callbacks argument that lets you specify a list of objects that Keras will call at the start and end of training, at the start and end of each epoch, and even before and after processing each batch. For example, the ModelCheckpoint callback saves checkpoints of your model at regular intervals during training, by default at the end of each epoch:

Using Callbacks

```
checkpoint_cb =
  keras.callbacks.ModelCheckpoint("my_keras_model.h5")
history = model.fit(X_train, y_train, epochs=10,
                    callbacks=[checkpoint_cb])
```

- If you use a validation set during training, you can set save_best_only=True when creating the ModelCheckpoint. In this case, it will only save your model when its performance on the validation set is the best so far. This way, you do not need to worry about training for too long and overfitting the training set: simply restore the last model saved after training, and this will be the best model on the validation set.
- Another way to implement early stopping is to simply use the EarlyStopping callback. It will interrupt training when it measures no progress on the validation set for a number of epochs (defined by the patience argument), and it will optionally roll back to the best model.

Using Callbacks

- You can combine both callbacks to save checkpoints of your model (in case your computer crashes) and interrupt training early when there is no more progress (to avoid wasting time and resources):

```
checkpoint_cb =
   keras.callbacks.ModelCheckpoint("my_keras_model.h5",
                                   save_best_only=True)
early_stopping_cb =
   keras.callbacks.EarlyStopping(patience=10,
                                 restore_best_weights=True)
history = model.fit(X_train, y_train, epochs=100,
                    validation_data=(X_valid, y_valid),
                    callbacks=[checkpoint_cb,
                               early_stopping_cb])
```

- The number of epochs can be set to a large value since training will stop automatically when there is no more progress.

Using Callbacks

- There is no need to restore the best model saved because the EarlyStopping callback will keep track of the best weights and restore them for you at the end of training
- If you need extra control, you can easily write your own custom callbacks. As an example of how to do that, the following custom callback will display the ratio between the validation loss and the training loss during training (e.g., to detect overfitting):

```
class PrintValTrainRatioCallback(keras.callbacks.Callback):
  def on_epoch_end(self, epoch, logs):
    ratio = logs["val_loss"] / logs["loss"])
    print(f"\nval/train: {ratio}")
```

Using Callbacks

- As you might expect, you can implement on_train_begin(), on_train_end(), on_epoch_begin(), on_epoch_end(), on_batch_begin(), and on_batch_end().
- Callbacks can also be used during evaluation and predictions, should you ever need them (e.g., for debugging).
- For evaluation, you should implement on_test_begin(), on_test_end(), on_test_batch_begin(), or on_test_batch_end() (called by evaluate())
- For prediction you should implement on_predict_begin(), on_predict_end(), on_predict_batch_begin(), or on_predict_batch_end() (called by predict()).

Using TensorBoard for Visualization

- TensorBoard is a great interactive visualization tool that you can use to view the learning curves during training, compare learning curves between multiple runs, visualize the computation graph, analyze training statistics, view images generated by your model, visualize complex multidimensional data projected down to 3D and automatically clustered for you, and more! This tool is installed automatically when you install TensorFlow
- To use it, you must modify your program so that it outputs the data you want to visualize to special binary log files called event files.
- Each binary data record is called a summary.
- The TensorBoard server will monitor the log directory, and it will automatically pick up the changes and update the visualizations: this allows you to visualize live data (with a short delay), such as the learning curves during training.

Using TensorBoard for Visualization

- In general, you want to point the TensorBoard server to a root log directory and configure your program so that it writes to a different subdirectory every time it runs. This way, the same TensorBoard server instance will allow you to visualize and compare data from multiple runs of your program, without getting everything mixed up.
- Cells 81-91, l5_n1.ipynb
- The TensorBoard() callback has options to log extra data too, such as embeddings
- Additionally, TensorFlow offers a lower-level API in the tf.summary package.

Fine-Tuning Neural Network Hyperparameters

- 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 simple MLP you can change the number of layers, the number of neurons per layer, the type of activation function to use in each layer, the weight initialization logic, and much more. How do you know what combination of hyperparameters is the best for your task?
- One option is to simply try many combinations of hyperparameters and see which one works best on the validation set (or use K-fold cross-validation). For example, we can use GridSearchCV or RandomizedSearchCV to explore the hyperparameter space
- Cells 92-105, l5_n1.ipynb
- Using randomized search is not too hard, and it works well for many fairly simple problems. When training is slow, however (e.g., for more complex problems with larger datasets), this approach will only explore a tiny portion of the hyperparameter space.

Fine-Tuning Neural Network Hyperparameters

- There are many techniques to explore a search space much more efficiently than randomly. Their core idea: when a region of the space turns out to be good, it should be explored more. Such techniques take care of the "zooming" process for you and lead to much better solutions in much less time. Here are some Python libraries you can use to optimize hyperparameters:
  Hyperopt, Hyperas, kopt, Talos, Keras Tuner, Scikit-Optimize (skopt), Spearmint, Hyperband, Sklearn-Deap
- Moreover, many companies offer services for hyperparameter optimization, for example, Google Cloud AI Platform's hyperparameter tuning service

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.
- But 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.
- Real-world data is often structured in such a hierarchical way, and deep neural networks automatically take advantage of this fact: lower hidden layers model low-level structures (line segments of various shapes and orientations), intermediate hidden layers combine these low-level structures to model intermediate-level structures (squares, circles), and the highest hidden layers and the output layer combine these intermediate structures to model high-level structures (faces)

Number of Hidden Layers

- Not only does this hierarchical architecture help DNNs converge faster to a good solution, but it also improves their ability to generalize to new datasets. For example, 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 kickstart 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

- In summary, for many problems you can start with just one or two hidden layers and the neural network will work just fine.
- For instance, you can easily reach above 97% accuracy on the MNIST dataset using just one hidden layer with a few hundred neurons, and above 98% accuracy using two hidden layers with the same total number of neurons, in roughly the same amount of training time.
- 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 or speech recognition, typically require networks with dozens 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 much more common to reuse parts of a pretrained 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. For example, the MNIST task requires $28 \times 28 = 784$ input neurons and 10 output neurons.
- These days one typically puts the same number of neurons in each hidden layer
- In practice, it's often simpler and more efficient to pick a model with more layers and neurons than you actually need, then use early stopping and other regularization techniques to prevent it from overfitting.
- In general you will get more bang for your buck by increasing the number of layers instead of the number of neurons per layer.

Learning Rate, Batch Size, and Other Hyperparameters

- Learning rate - the optimal learning rate is about half of the maximum learning rate - the learning rate above which the training algorithm diverges
- Optimizer - there are other optimizers besides Mini-batch Gradient Descent and we shall consider them next time
- Batch size - can have a significant impact on your model's performance and training time. The main benefit of using large batch sizes is that hardware accelerators like GPUs can process them efficiently, so the training algorithm will see more instances per second. On the other hand, for large batch sizes your data might not fit into GPU memory. Also large batch sizes often lead to training instabilities, especially at the beginning of training, and the resulting model may not generalize as well as a model trained with a small batch size.

Learning Rate, Batch Size, and Other Hyperparameters

- Activation function - in general, the ReLU activation function will be a good default for all hidden layers. For the output layer, it really depends on your task.
- Number of iterations - in most cases does not actually need to be tweaked: just use early stopping instead.