Introduction to Neural Networks

@Phi Vu Tran

Tutorial Overview

In this tutorial, we use the Python programming language to implement a simple 3-layer feedforward neural network from scratch to solve a multi-classification problem. Neural networks are often mystified as "black box" engines that just seem to work without much intuition. The goal of this exercise is to see how neural networks work under the hood, specifically the main ideas of backpropagation and gradient descent, and learn how to adapt and apply these concepts to new problems.

In order to bound the scope of this tutorial, we assume some basic knowledge of Python coding syntax and supervised machine learning concepts such as logistic regression. Although we strive to incorporate many theoretical and mathematical details to help facilitate the topic, the reader is encouraged to expand upon this tutorial with further reading and research. This tutorial highlights the following important concepts commonly found in a typical machine learning workflow:

- Cross-validation;
- Model training, evaluation, and visualization;
- Hyperparameter selection for experimental design;
- Regularization techniques to mitigate overfitting.

We hope that this tutorial supplements your understanding of deep learning and encourages you to seek the available resources on deep learning frameworks (TensorFlow, Keras, PyTorch, Caffe(2), MXNet) to facilitate the design and implementation of useful models to advance your own analytical objectives.

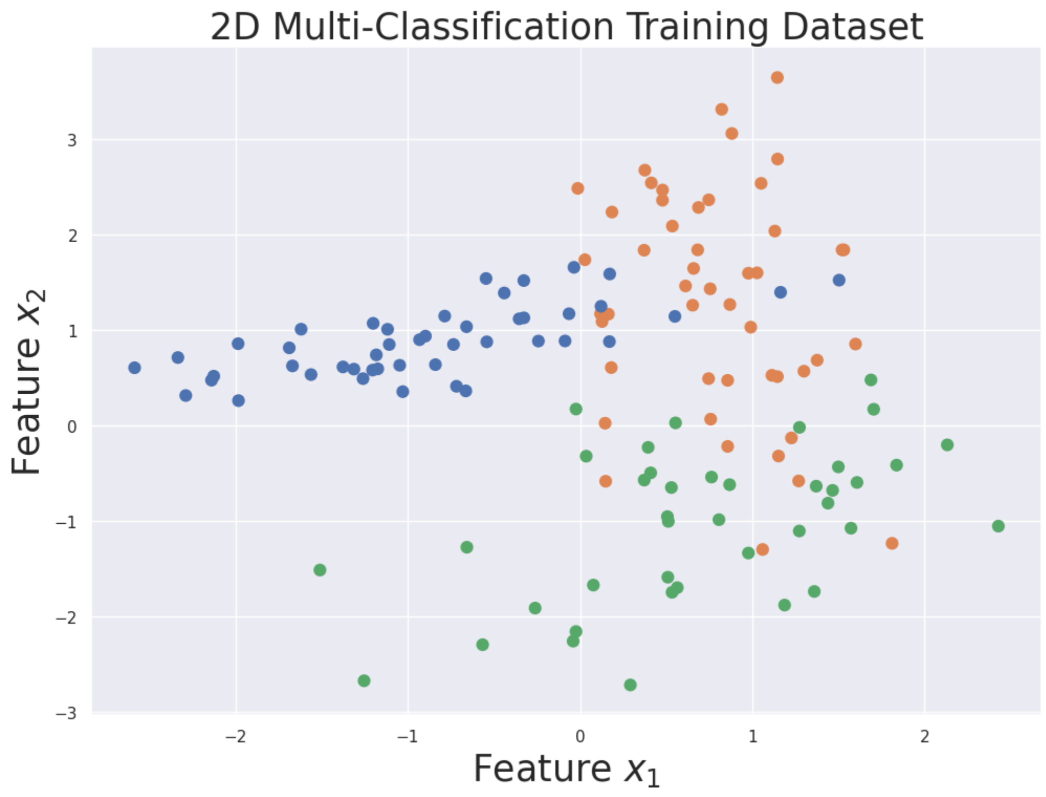
Acknowledgment - we build upon and expand the tutorials previously made available here and here. The reader is encouraged to explore more online resources related to neural networks and deep learning in the Further Reading section.

```
# Package imports
import matplotlib.pyplot as plt
from matplotlib.colors import ListedColormap
import seaborn as sns
sns.set()
sns.set_style('darkgrid')
import numpy as np
from sklearn.datasets import make_classification
from sklearn.linear_model import LogisticRegressionCV
from sklearn.model_selection import train_test_split
from sklearn.metrics import classification_report
from itertools import chain
```

1. Generate Dataset

We start by generating a dataset to demonstrate supervised machine learning. Every machine learning study and experiment hinges on some training dataset, and obtaining quality data can be a non-starter. Fortunately, scikit-learn has some useful dataset generators and loaders to help us get started. For this tutorial, we select the `make_classification` module to create a 2D multi-classification toy dataset.

```
X, y = make_classification(n_samples=200, n_features=2, n_informative=2,
                           n_redundant=0, n_repeated=0, n_classes=3, n_clusters_per_class=1,
                           weights=None, flip_y=0.01, class_sep=0.85, hypercube=True, shift=0.0,
                           scale=1.0, shuffle=True, random_state=0)

# Split the dataset into training and validation segments
X, X_val, y, y_val = train_test_split(X, y, test_size=0.33, random_state=1234)
# Define our color palette
palette = sns.color_palette('deep', 3)
colormap = ListedColormap(palette) # colormap follows the BGR palette
# Plot training dataset
plt.rcParams['figure.figsize']=(11.7, 8.27)
fontsize = 25
plt.scatter(X[:, 0], X[:, 1], s=60, c=y, cmap=colormap)
plt.xlabel('Feature $x_1$', fontsize=fontsize)
plt.ylabel('Feature $x_2$', fontsize=fontsize)
plt.title('2D Multi-Classification Training Dataset', fontsize=fontsize)
plt.show()
```



The dataset we generated has three classes, plotted as blue, green, and red circles. Programmatically, the three classes are represented as zero-based indices. For our dataset, class 0 is represented by blue circles, class 1 by green circles, and class 2 by red circles. The colors of the circles can be interpreted as real-world categories such as different animal species, and the x- and y-coordinates being some measureable features about each species.

By design, our dataset only has two features, which are easy to visualize. Real-world datasets may have hundreds or thousands of features over many categories, making the task of statistical learning from data very difficult. Thus, we emphasize the importance of the scientific method and sound experimental design to build robust models that produce meaningful and reliable results.

We also split the dataset into training and validation segments, as is customary in a typical machine learning experimentation setup. We train and tune our model on the training segment, and validate the model on the validation set based on some performance metrics. This is known in the literature as cross-validation.

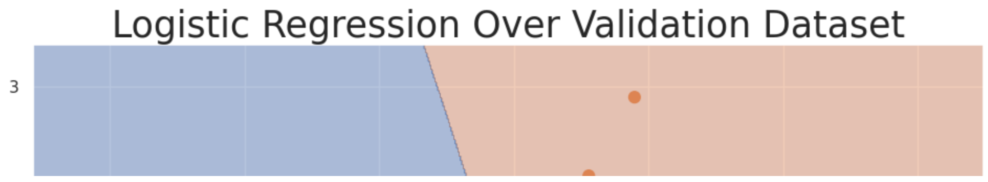
2. Logistic Regression

We baseline our machine learning experiments with logistic regression, and then compare its accuracy performance against a simple feedforward neural network. Logistic regression is a linear probabilistic classifier that works well in many classification tasks because of its proven high performance, ease of implementation, and simplicity in interpretation. The input to the logistic regression model is some training dataset consisting of x observable feature values and corresponding y ground-truth labels. Logistic regression outputs well-calibrated probabilities that can be thresholded to give the predicted class labels. For convenience, we use the logistic regression module from `scikit-learn` to fit our training dataset and visualize its decision boundary. For multi-classification settings where there are more than two classes, we choose `multinomial` over the default `one vs. rest` learning scheme.

```
# Train the logistic regression linear classifier
clf = LogisticRegressionCV(Cs=10, class_weight=None, cv=None, dual=False,
                           fit_intercept=True, intercept_scaling=1.0, max_iter=200,
                           multi_class='multinomial', n_jobs=1, penalty='l2', random_state=0,
                           refit=True, scoring=None, solver='sag', tol=0.0001, verbose=0)
clf.fit(X, y)
```

```
                         LogisticRegressionCV
LogisticRegressionCV(max_iter=200, multi_class='multinomial', n_jobs=1,
                     random_state=0, solver='sag')
```

```
def plot_decision_boundary(pred_func):
    """
    Helper function to plot the decision boundary as a contour plot.

    Argument
    --------
    pred_func : function that takes a trained model's parameters (weights)
        and produces an output vector of predicted class labels over a dataset
    """

    # Set min and max values and give it some padding
    x_min, x_max = X_val[:, 0].min() - 0.5, X_val[:, 0].max() + 0.5
    y_min, y_max = X_val[:, 1].min() - 0.5, X_val[:, 1].max() + 0.5
    h = 0.01
    # Generate a grid of points with distance h between them
    xx, yy = np.meshgrid(np.arange(x_min, x_max, h),
                         np.arange(y_min, y_max, h))
    # Plot the function decision boundary. For that, we will assign
    # a color to each point in the mesh [x_min, x_max] by [y_min, y_max].
    Z = pred_func(np.c_[xx.ravel(), yy.ravel()])
    Z = Z.reshape(xx.shape)
    # Plot the contour and training examples
    plt.contourf(xx, yy, Z, cmap=colormap, alpha=0.4)
    plt.scatter(X_val[:, 0], X_val[:, 1], s=60, c=y_val, cmap=colormap)

# Plot the decision boundary.
# First define the prediction function, which takes a trained model's weights (parameters)
# to produce an output vector of predicted integer classes over a dataset
pred_func = lambda x: clf.predict(x)
plot_decision_boundary(pred_func)
plt.xlabel('Feature $x_1$', fontsize=fontsize)
plt.ylabel('Feature $x_2$', fontsize=fontsize)
plt.title('Logistic Regression Over Validation Dataset', fontsize=fontsize)
plt.show()
```



The plot above shows the decision boundary learned by the logistic classifier. Qualitatively, it looks like the logistic classifier does a good job at separating the data into the three classes using straight lines. But quantitatively, how do we assess the model's classification performance? We will take a closer look at model evaluation in the Numerical Experiments section.

3. Feedforward Neural Network

We now switch our attention to building and characterizing a simple 3-layer feedforward neural network, also known as a multi-layer perceptron, that has one input layer, one hidden layer, and one output layer. Like the name implies, the information flow in feedforward neural networks does not form a cycle, and feedforward network architectures can form arbitrary directed acyclic graphs (DAGs).

The input to the network is a vector of features x_1 and x_2, along with a *bias unit* b that can be interpreted as the intercept term to help fit data patterns better. The number of units in the input layer is 2, corresponding to the dimensionality of the input data, not counting the bias unit. The number of units in the output layer is 3, corresponding to the number of classes. The output of the network is the probability distribution over class labels. The schematic of the neural network looks something like this:

Neural Network Diagram

The dimensionality of the hidden layer is typically set as a user-defined *hyperparameter*. The more units we put into the hidden layer, the more complex functions the network is able to fit. But higher dimensionality comes at a cost for two reasons: (1) more computation is required to learn the network parameters and make predictions; and (2) a larger number of parameters is more prone to overfitting, resulting in a model with poor predictive performance.

How to choose the size of the hidden layer? While there are some general guidelines and recommendations, one way to determine the size of the hidden layer is through cross-validation. In the Numerical Experiments section, we explore how the size of the hidden layer impacts the accuracy performance of the network on a validation dataset.

3.1 Forward Propagation

The network makes predictions using *forward propagation*, which is just a bunch of matrix multiplications followed by a bias offset and then element-wise application of an *activation function*. For a more compact notation, we drop the superscripts and subscripts and adopt the matrix-vectorial notation. If x is the 2-dimensional input to the network, then we compute the 3-dimensional prediction $\hat{y}$ as follows:

$$\begin{aligned} z^{(2)} &= xW^{(1)} + b^{(1)} \\ a^{(2)} &= f\left(z^{(2)}\right) \\ z^{(3)} &= a^{(2)}W^{(2)} + b^{(2)} \\ a^{(3)} &= \hat{y} = \mathrm{softmax}\left(z^{(3)}\right) \end{aligned}$$

$z^{(i)}$ is the input of layer i and $a^{(i)}$ is the output of layer i after applying the activation function $f(z)$. $W^{(1)}, b^{(1)}, W^{(2)}, b^{(2)}$ are the parameters of the network, which we need to learn from the training data.

$z^{(2)}$ is the dot (inner) product of the input feature vector x and the weight matrix $W^{(1)}$, offset by the bias vector $b^{(1)}$. Similarly, $z^{(3)}$ is the inner product of the activation vector $a^{(2)}$ and the weight matrix $W^{(2)}$, offset by the bias vector $b^{(2)}$. Recalling the algebraic definition of the dot product, and assuming we use 500 units for the hidden layer, then the dimensionalities of the weight matrices and bias vectors are: $W^{(1)} \in \mathbb{R}^{2\times 500}$, $b^{(1)} \in \mathbb{R}^{500}$, $W^{(2)} \in \mathbb{R}^{500\times 3}$, $b^{(2)} \in \mathbb{R}^{3}$

3.2 Activation Functions

The units of the hidden layer $a^{(2)}$ transform the input features through an element-wise non-linear *activation function*, which is critical for the neural network to model and fit non-linear hypotheses in the data. Common choices for the activation function are the sigmoid function, $\sigma(z)$, the hyperbolic tangent function, $\tanh(z)$, and the rectified linear unit, $\mathrm{relu}(z)$:

$$\sigma(z) = \frac{1}{1+e^{-z}}$$
$$\tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$$
$$\text{relu}(z) = \max(0, z)$$

We use the ReLU activation function in this tutorial, which performs very well in many scenarios. It also has some nice non-linearity benefits such as:

- Induce sparsity of feature activations;
- Simplify backpropagation because the derivative of ReLU is simply:

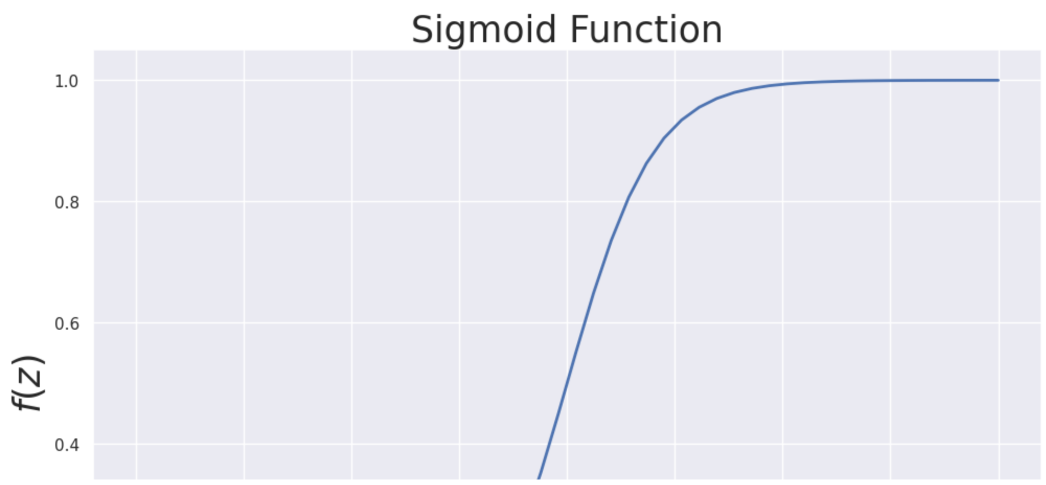
$$\text{relu}'(z) = \frac{d\text{relu}(z)}{dz} = \begin{cases} 1 & z > 0 \\ 0 & z \leq 0 \end{cases}$$

- Avoid saturation and vanishing gradient issues associated with sigmoid or tanh;
- More efficient learning and better convergence even without pre-training.

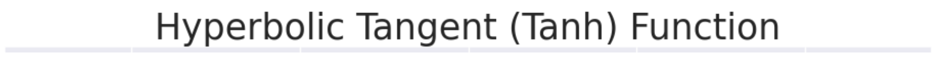
Similarly, the units of the output layer $a^{(3)}$ transform the units of the hidden layer through the element-wise non-linear activation function softmax. The softmax function is used at the output layer as a probabilistic classifier that outputs normalized, well-calibrated class probabilities. The softmax classifier is the generalization of the logistic classifier to multiple classes.

Except for the bias units, all other units in this network architecture are densely, or fully, connected to each other. The downside of each unit being *independently* fully connected to all other units is the large number of parameters needed to learn from the data. Consider an example where the input is an image with the modest size of 28x28 pixels, and the hidden layer has dimensionality of 100, the amount of parameters to learn is on the order of 10^5. Now suppose the input image has a larger resolution of 96x96 pixels, then the amount of parameters to learn is on the order of 10^6. The combined forward and backward propagation computations would be about $10 \times 10 = 100$ times slower for the 96x96 image compared to the 28x28 image. A typical image of 256x256 resolution then becomes computationally prohibitive when using a fully connected network.

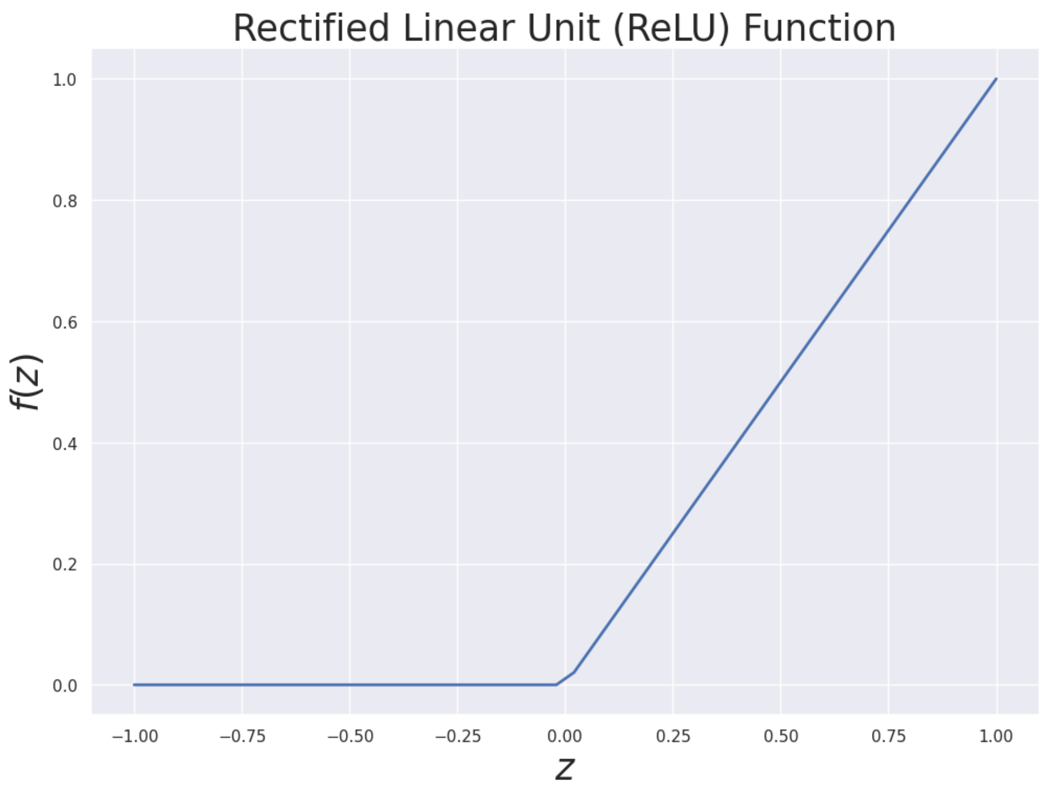
```
def sigmoid(z):
    """Helper function to compute the element-wise
    sigmoid activation of an array.
    """

    sigmoid = 1.0 / (1.0 + np.exp(-z))
    dsigmoid = sigmoid * (1.0 - sigmoid)

    return sigmoid, dsigmoid

# Plot example sigmoid function
z = np.linspace(-10,10)
fz, _ = sigmoid(z)
plt.plot(z, fz, lw=2)
plt.xlabel('$z$', fontsize=fontsize)
plt.ylabel('$f(z)$', fontsize=fontsize)
plt.title('Sigmoid Function', fontsize=fontsize)
plt.show()
```



```
def tanh(x):
    """Helper function to compute the element-wise
    hyperbolic tangent activation of an array.
    """

    tanh = np.tanh(x)
    dtanh = 1.0 - np.square(tanh)

    return tanh, dtanh

# Plot example tanh function
z = np.linspace(-5,5)
fz, _ = tanh(z)
plt.plot(z, fz, lw=2)
plt.xlabel('$z$', fontsize=fontsize)
plt.ylabel('$f(z)$', fontsize=fontsize)
plt.title('Hyperbolic Tangent (Tanh) Function', fontsize=fontsize)
plt.show()
```



```
def relu(z):
    """
    Helper function to compute the element-wise ReLU activation
    of an array by simple and efficient thresholding.
    """

    relu = z * (z > 0)
    drelu = 1.0 * (z > 0)

    return relu, drelu

# Plot example ReLU function
z = np.linspace(-1.0, 1.0)
fz, _ = relu(z)
plt.plot(z, fz, lw=2)
plt.xlabel('$z$', fontsize=fontsize)
plt.ylabel('$f(z)$', fontsize=fontsize)
plt.title('Rectified Linear Unit (ReLU) Function', fontsize=fontsize)
plt.show()
```



Putting everything together, along with some helper functions, here is how we compute the forward propagation with ReLU non-linearity.

$$
\begin{aligned}
z^{(2)} &= xW^{(1)} + b^{(1)} \\
a^{(2)} &= \mathrm{relu}\left(z^{(2)}\right) \\
z^{(3)} &= a^{(2)}W^{(2)} + b^{(2)} \\
a^{(3)} &= \hat{y} = \mathrm{softmax}\left(z^{(3)}\right)
\end{aligned}
$$

```
def softmax(z):
    """Helper function to compute the element-wise softmax activation of an array."""
    # Shift argument to prevent potential numerical instability from large exponentials
    exp = np.exp(z - np.max(z))
    probs = exp / np.sum(exp, axis=1, keepdims=True)

    return probs


def reshape(array):
    """Helper function to reshape input array into weight matrices and bias vectors."""

    W1 = np.reshape(array[0:input_dim * hidden_dim],
                    (input_dim, hidden_dim))
    W2 = np.reshape(array[input_dim * hidden_dim:hidden_dim * (input_dim + output_dim)],
                    (hidden_dim, output_dim))
    b1 = array[hidden_dim * (input_dim + output_dim):hidden_dim * (input_dim + output_dim + 1)]
    b2 = array[hidden_dim * (input_dim + output_dim + 1):]

    return W1, W2, b1, b2


def forward(params, x, predict=False):
    """The forward pass to predict softmax probability distribution over class labels."""

    W1, W2, b1, b2 = reshape(params)

    # Forward propagation
    z2 = np.dot(x, W1) + b1
    a2, _ = relu(z2)
    z3 = np.dot(a2, W2) + b2
    probs = softmax(z3)

    return np.argmax(probs, axis=1) if predict else probs
```

3.3 Learning and Optimization

When we train the neural network to learn the underlying patterns, signatures, or features of some dataset, we are finding parameters ($W^{(1)}, b^{(1)}, W^{(2)}, b^{(2)}$) that minimize the errors between the network predictions $\hat{y}$ and the ground-truths y. But how do we define the error? We call the function that measures error the *loss function*, or sometimes referred to as the *cost function* or the *objective*. Hence, the learning task is defined by the loss function. For the task of multi-classification with the softmax output, the learning function is the cross-entropy loss. If we have N training examples and C classes, then the total loss for the network predictions $\hat{y}$ with respect to the true labels y is given by:

$$L(y, \hat{y}) = \underbrace{-\frac{1}{N}\sum_{n \in N}\sum_{c \in C} y_{n,c} \log \hat{y}_{n,c}}_{\text{data loss}} + \underbrace{\frac{\lambda}{2}\sum_{l}\sum_{j}\sum_{k}\left(W_{j,k}^{(l)}\right)^2}_{\text{regularization penalty loss}}$$

The formula for the total loss $L(y, \hat{y})$ comprises two component losses: data loss and regularization penalty loss. The data loss is the function of the difference between the estimated value $\hat{y}$ and true value y for an instance of the training example. For a fixed training set of examples, the data loss becomes the average error over all examples. In essence, the further away $\hat{y}$ and y are, the greater the data loss.

For large and complex networks, the parameters W may be linearly related and their values are not well determined. In other words, the set of parameters W is not necessarily unique, and their values can be strongly influenced by randomness in the training data. The ambiguity of the parameters and their linear dependence is a main reason for overfitting. To help overcome overfitting, we implement a regularization function that explicitly penalizes large values of the parameters to reduce the influence of linear dependence.

The most common regularization penalty is the L_2 norm that discourages large weights through an element-wise quadratic penalty over all parameters W. In the regularization penalty expression, we sum up all the squared elements of W. Notice that the regularization penalty is only a function of the parameters W and not of the data. Therefore, the regularization penalty is often referred to as *weight decay*. The hyperparameter λ controls the strength of regularization, with small values of λ strongly penalizing large weights. The optimal value of λ is dataset-specific, and is usually determined by cross-validation. Including the regularization penalty in the total loss expression is optional, and can help control overfitting in many cases. The total loss function quantifies our unhappiness with predictions on the training set.

The loss function is generally unknown because it is a function of the parameters ($W^{(1)}, b^{(1)}, W^{(2)}, b^{(2)}$), which are to be learned from data. Recent work from Li et al., 2018 produced meaningful visualizations of various high-dimensional non-convex neural loss functions and showed that the resulting *loss landscapes* are incredibly complex, with sharp and flat local geometries that directly impact learning and generalization of neural networks. For some initial values of the parameters, computing the loss function is relatively straightforward via forward propagation and comparison of the errors with the ground-truths. The main challenge is to iteratively update our initial guess so that we find the right set of parameters that converge to the minimum of the loss landscape.

Loss Landscape

Source: Li et al., 2018. https://arxiv.org/abs/1712.09913

```
def compute_loss(params):
    """ Function to compute the average loss over the dataset. """

    W1, W2, b1, b2 = reshape(params)

    # Forward propagation
    probs = forward(params, X, predict=False)

    # Compute the loss
    correct_logprobs = np.log(probs[range(num_examples), y])
    data_loss = -np.sum(correct_logprobs) / num_examples # data loss

    reg_loss = weight_decay/2.0 * \
            (np.sum(np.square(W1)) + np.sum(np.square(W2))) # weight_decay is lambda

    # Optionally add weight decay regularization term to loss
    loss = data_loss + reg_loss  # total loss including regularization

    return loss
```

3.4 Backward Propagation (Backprop)

Recall that the goal is to learn the parameters that minimize the loss function. Hence, learning is cast as an optimization problem. Gradient descent is used to minimize the loss function by taking iterative steps in the direction of the negative gradient. Although gradient descent is susceptible to finding local minima of many non-convex loss functions, the algorithm usually works fairly well in practice.

Like the name implies, gradient descent needs the gradient (vector of derivatives) of the loss function with respect to the parameters (W, b): $\frac{\partial L}{\partial W^{(1)}}, \frac{\partial L}{\partial b^{(1)}}, \frac{\partial L}{\partial W^{(2)}}, \frac{\partial L}{\partial b^{(2)}}$. To compute the gradient, we use the seminal *backpropagation algorithm*, which repeatedly applies the chain rule to differentiate compositions of functions. The mathematical details behind the backpropagation algorithm are beyond the scope of this tutorial; the reader is referred to some excellent expositions of the backpropagtion algorithm found here, here, and here.

Applying backpropagation algorithm to the network, we find the following expressions for computing the desired gradient at each layer of the network:

$$\begin{aligned}
\delta^{(3)} &= \frac{\partial L}{\partial z^{(3)}} = \hat{y} - y \\
\delta^{(2)} &= \delta^{(3)} (W^{(2)})^T \circ \mathrm{relu}'(z^{(2)}) \\
\frac{\partial L}{\partial W^{(2)}} &= (a^{(2)})^T \delta^{(3)} \\
\frac{\partial L}{\partial b^{(2)}} &= \delta^{(3)} \\
\frac{\partial L}{\partial W^{(1)}} &= x^T \delta^{(2)} \\
\frac{\partial L}{\partial b^{(1)}} &= \delta^{(2)}
\end{aligned}$$

Now that we have the expressions to compute the gradient of the loss function at each layer of the network, we can train the neural network by taking repeated steps in the direction of the *negative* gradient to *reduce* error in the loss function. The *vanilla* gradient descent procedure for performing a parameter update is as follows:

$$\begin{aligned}
W^{(l)} &\leftarrow W^{(l)} - \alpha \left[\frac{1}{N} \frac{\partial L}{\partial W^{(l)}} + \lambda W^{(l)} \right], \\
b^{(l)} &\leftarrow b^{(l)} - \alpha \left[\frac{1}{N} \frac{\partial L}{\partial b^{(l)}} \right],
\end{aligned}$$

where the supersript (l) refers to the layer index, α is the step size (or learning rate), N is the number of training examples, $\left(\frac{\partial L}{\partial W^{(l)}}, \frac{\partial L}{\partial b^{(l)}}\right)$ are the gradient of the loss function with respect to parameters (W, b) at layer l, and λ is the strength of regularization (or weight decay).

The negative gradient tells us the direction in which the function has the steepest rate of decrease, but it does not tell us how far along this direction we should step. The step size α is an important hyperparameter that can directly impact network performance by influencing how quickly gradient descent converges to a local minimum, or even diverges. The step size must be carefully tuned and set *just right*. If it is set too low, the training progress is steady but slow. And if it is set too high, the training progress can be faster, but more at risk of diverging to the land of `NaN`.

One best practice to tuning the learning rate α is to initially set it to a relatively high value like `lr = 0.1` to stall the network resulting in large loss, and then successively halve the learning rate until the network begins to show signs of convergence. This procedure may take a few tries to properly work, but it is one way to ensure the network gets the sub-optimal value of α for peak performance.

Putting everything together, the Python code below trains the neural network by way of implementing the concepts of forward propagation, backpropagation, and gradient descent over a number of passes on the training set, or epochs.

```
def train_nn(hidden_dim, num_passes=200, update_params=True, dropout_ratio=None, print_loss=None):
    """
    This function learns parameters for the neural network via backprop and batch gradient descent.

    Arguments
    ----------
    hidden_dim : Number of units in the hidden layer
    num_passes : Number of passes through the training data for gradient descent
    update_params : If True, update parameters via gradient descent
    dropout_ratio : Percentage of units to drop out
    print_loss : If integer, print the loss every integer iterations

    Returns
    -------
    params : updated model parameters (weights) stored as an unrolled vector
    grad : gradient computed from backprop stored as an unrolled vector
    """

    # Initialize the parameters to random values. We need to learn these.
    np.random.seed(1234)
    W1 = np.random.randn(input_dim, hidden_dim) * np.sqrt(2.0 / input_dim)
    W2 = np.random.randn(hidden_dim, output_dim) * np.sqrt(2.0 / hidden_dim)
    b1 = np.ones((1, hidden_dim))
    b2 = np.ones((1, output_dim))

    # Gradient descent. For each batch...
    for i in range(0, num_passes):

        # Forward propagation
        z2 = np.dot(X, W1) + b1
        a2, da2 = relu(z2)
        if dropout_ratio is not None:
            # Perform inverted dropout.
            # See more info at http://cs231n.github.io/neural-networks-2/#reg
            p = 1.0 - dropout_ratio # Probability of dropping a unit
            u2 = (np.random.rand(*a2.shape) < p) / p
            a2 *= u2
        z3 = np.dot(a2, W2) + b2
        probs = softmax(z3)

        # Back propagation
        delta3 = probs
        delta3[range(num_examples), y] -= 1.0
        delta2 = np.dot(delta3, W2.T) * da2
        if dropout_ratio is not None:
            # Perform inverted dropout.
            # See more info at http://cs231n.github.io/neural-networks-2/#reg
            delta2 *= u2
        dW2 = np.dot(a2.T, delta3)
        db2 = delta3
        dW1 = np.dot(X.T, delta2)
        db1 = delta2

        # Scale gradient by the number of examples and add regularization to the weight terms.
        # We do not regularize the bias terms.
        dW2 = (dW2/num_examples) + weight_decay * W2
        dW1 = (dW1/num_examples) + weight_decay * W1
        db2 = np.sum(db2, axis=0, keepdims=True) / num_examples
        db1 = np.sum(db1, axis=0, keepdims=True) / num_examples

        if update_params:
            # Gradient descent parameter update
            W1 += -lr * dW1
            b1 += -lr * db1
            W2 += -lr * dW2
            b2 += -lr * db2

        # Unroll model parameters and gradient and store them as a long vector
        params = np.asarray(list(chain(*[W1.flatten(),
                                         W2.flatten(),
                                         b1.flatten(),
                                         b2.flatten()
                                        ]
                                      )
                                )
                           )
        grad = np.asarray(list(chain(*[dW1.flatten(),
                                       dW2.flatten(),
                                       db1.flatten(),
                                       db2.flatten()
                                      ]
                                    )
                              )
                         )

        # Optionally print the loss after some number of iterations
        if (print_loss is not None) and (i % print_loss == 0):
            print('Loss after iteration %i: %f' %(i, compute_loss(params)))

    return params, grad
```

3.5 Gradient Verification

Backpropagation is a notoriously difficult algorithm to debug and get right because there are many subtle and nuanced details associated with it. Then, how do we know if the mathematical derivations of the gradient, and their code implementations, are correct? A buggy backpropagation implementation may still produce surprisingly reasonable results, while performing less well than the correct implementation. Gradient descent still steps in the direction of the (wrong) negative gradient to reduce error in the loss function, but does not reach a local minimum if the gradient is incorrect. In this section, we utilize a procedure for numerically checking the computed gradient to ensure correctness in our math and code.

Recall the centered finite difference formula for computing the numerical derivative of a univariate function is:

$$\frac{df(x)}{dx} = \lim_{\epsilon \to 0} \frac{f(x+\epsilon) - f(x-\epsilon)}{2\epsilon}$$

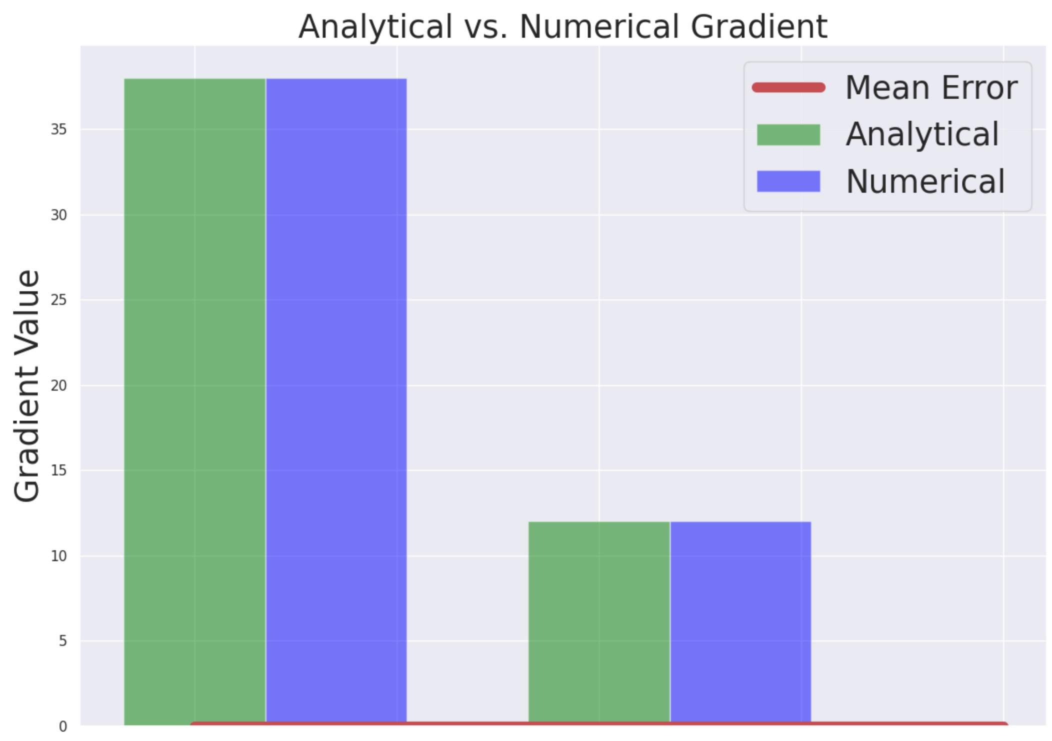
The above expression is also called the symmetric derivative, and it can be used to approximate the derivative of any arbitrary differentiable function. For a function of multiple variables, the gradient is the vector of *partial derivatives* with respect to each variable. In the context of our neural network, the function of interest is the loss function $L\left(y, \hat{y}\right)$ and the variables are the parameters (W, b). Choosing ϵ to be a small constant, say 10^{-4}, we utilize the symmetric derivative to approximate the numerical gradient of $L\left(y, \hat{y}\right)$ as shown in the Python code snippet below.

```
def compute_numerical_gradient(func, params):
    """
    For an arbitrary differentiable vector-valued function,
    compute the numerical gradient of the function using the symmetric derivative.
    """

    eps = 1e-4
    num_grad = np.zeros(len(params))
    E = np.eye(len(params))
    for i in range(len(params)):
        params_plus  = params + eps * E[:, i]
        params_minus = params - eps * E[:, i]
        num_grad[i] = (func(params_plus) - func(params_minus)) / (2.0 * eps)

    return num_grad
```

Next, we verify the correctness of `compute_numerical_gradient()` using a simple quadratic function $f(x, y) = x^2 + 3xy$, which has the closed-form analytical gradient: $\frac{\partial f(x,y)}{\partial x} = 2x + 3y$, $\frac{\partial f(x,y)}{\partial y} = 3x$.

```
# Verify the correctness of compute_numerical_gradient() implementation
def quadratic_function(x):
    """
    Simple quadratic function that accepts 2D vector as input:
    f(x,y) = x^2 + 3xy
    """

    return x[0]**2 + 3*x[0]*x[1]

def quadratic_function_prime(x):
    """
    Compute the analytical gradient of f(x,y) = x^2 + 3xy to give 2D vector of partial derivatives
    df/dx = 2x + 3y
    df/dy = 3x
    """

    grad = np.zeros(x.shape)
    grad[0] = 2*x[0] + 3*x[1]
    grad[1] = 3*x[0]

    return grad

def plot_gradients(analytical_grad, numerical_grad, label1, label2):
    """
    Inspect analytical gradient and numerical gradient computed from centered finite difference,
    side by side
    """
    error_thresh = np.mean(np.abs(grad - num_grad))

    fig, ax = plt.subplots()
    bar_width = 0.35
    opacity = 0.5
    error_config = {'ecolor': '0.3'}

    index = np.arange(len(grad)+1)

    # Plot bar charts
    rects1 = ax.bar(index[:-1], analytical_grad, bar_width,
                    alpha=opacity, color='green',
                    error_kw=error_config, label=label1)

    rects2 = ax.bar(index[:-1] + bar_width, numerical_grad, bar_width,
                    alpha=opacity, color='blue',
                    error_kw=error_config, label=label2)

    # Plot horizontal line indicating error threshold
    ax.plot(index, np.repeat(error_thresh, repeats=len(grad)+1),
                             "r-", linewidth=8, label='Mean Error')

    ax.set_ylabel('Gradient Value', fontsize=fontsize)
    ax.set_title(' '.join([label1, 'vs.', label2, 'Gradient']), fontsize=fontsize)
    ax.legend(fontsize=fontsize)
    plt.tick_params(
        axis='x',          # changes apply to the x-axis
        which='both',      # both major and minor ticks are affected
        bottom=False,      # ticks along the bottom edge are off
        top=False,         # ticks along the top edge are off
        labelbottom=False  # labels along the bottom edge are off
    )
    fig.tight_layout()

x = np.array([4.0, 10.0]) # 2D vector input

grad = quadratic_function_prime(x) # analytical gradient for x
num_grad = compute_numerical_gradient(quadratic_function, x) # numerical gradient for x
abs_error = np.abs(grad - num_grad) # absolute error between grad and num_grad

plot_gradients(grad, num_grad, 'Analytical', 'Numerical')

print('\nMean Absolute Error: {:.12e}'.format(np.mean(abs_error)))
```

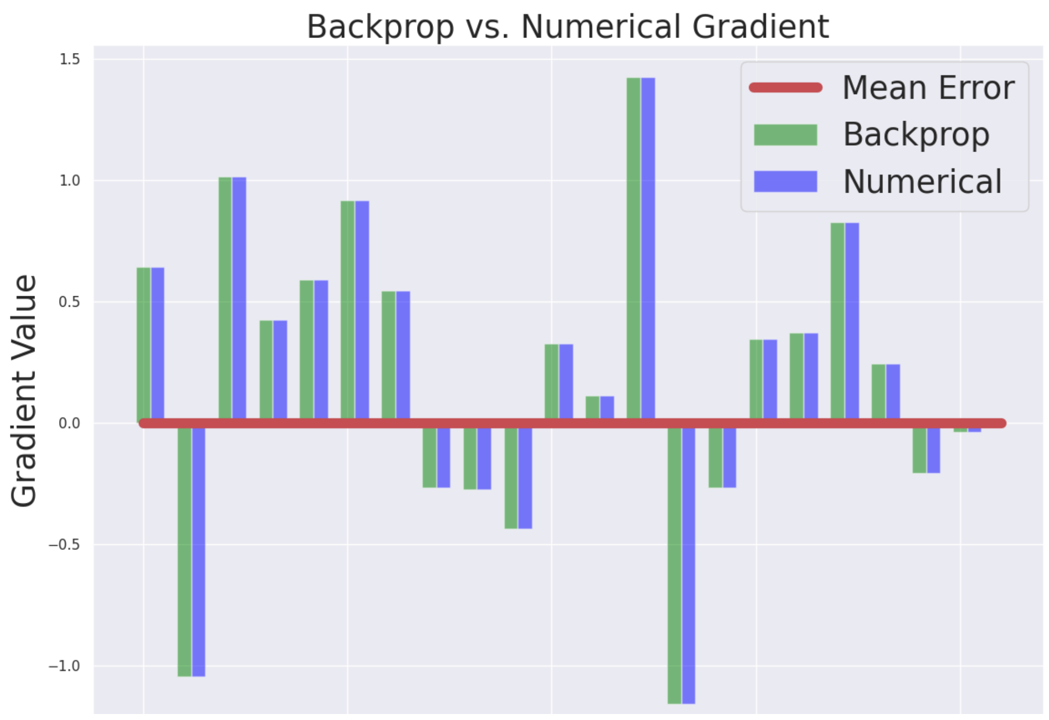
```
Mean Absolute Error: 9.379164112033e-11
```



Finally, we perform numerical gradient verification on the actual neural network with 3 units in the hidden layer. Feel free to experiment with various constants for the dimensionality of the hidden layer.

```
# perform numerical gradient checking of the neural network implementation
num_examples = X.shape[0] # number of examples
input_dim = X.shape[1] # input layer dimensionality
hidden_dim = 3 # hidden layer dimensionality
output_dim = len(np.unique(y)) # output layer dimensionality

lr = 0.01 # learning rate for gradient descent
weight_decay = 0.0005 # regularization strength, smaller values provide greater strength

params, grad = train_nn(hidden_dim, update_params=False, dropout_ratio=None, num_passes=1)
num_grad = compute_numerical_gradient(compute_loss, params)
abs_error = np.abs(grad - num_grad)

plot_gradients(grad, num_grad, 'Backprop', 'Numerical')

print('\nMean Absolute Error: {:.12e}'.format(np.mean(abs_error)))
```

```
Mean Absolute Error: 8.950444320884e-11
```



4. Numerical Experiments

In this section, we conduct some experiments to assess how the size or dimensionality of the hidden layer of the neural network can directly impact accuracy performance. We also demonstrate how the size of the hidden layer can result in network overfitting, and then utilize dropout regularization to mitigate overfitting.

Evaluation of Logistic Regression

```
# Evaluate logistic classifier on validation set
clf_preds = clf.predict(X_val)

print('\nClassification Report for Logistic Regression:\n')
print(classification_report(y_val, clf_preds))
print('Validation Accuracy for Logistic Regression: {:.6f}'.format(np.mean(y_val==clf_preds)))
```

```
Classification Report for Logistic Regression:

              precision    recall  f1-score   support

           0       0.89      0.85      0.87        20
           1       0.50      0.65      0.57        20
           2       0.67      0.54      0.60        26

    accuracy                           0.67        66
   macro avg       0.69      0.68      0.68        66
weighted avg       0.69      0.67      0.67        66

Validation Accuracy for Logistic Regression: 0.666667
```

Evaluation of Neural Network with 3 Hidden Units

```
# Fit a model with a 3-dimensional hidden layer
hidden_dim = 3

# Gradient descent parameters
lr = 0.01 # learning rate for gradient descent
weight_decay = 0.0005 # regularization strength, smaller values provide greater strength

# Fit neural network
params, _ = train_nn(hidden_dim, num_passes=20000, dropout_ratio=None, print_loss=1000)
```

```
# Evaluate accuracy performance on validation set
nn_preds = forward(params, X_val, predict=True)

print('\nClassification Report for Neural Network:\n')
print(classification_report(y_val, nn_preds))
print('Validation Accuracy for Neural Network: {:.6f}'.format(np.mean(y_val==nn_preds)))

# Plot the decision boundary
pred_func = lambda x: forward(params, x, predict=True)
plot_decision_boundary(pred_func)
plt.title('Decision Boundary for Hidden Layer Size {:d}'.format(hidden_dim), fontsize=fontsize)
plt.show()
```

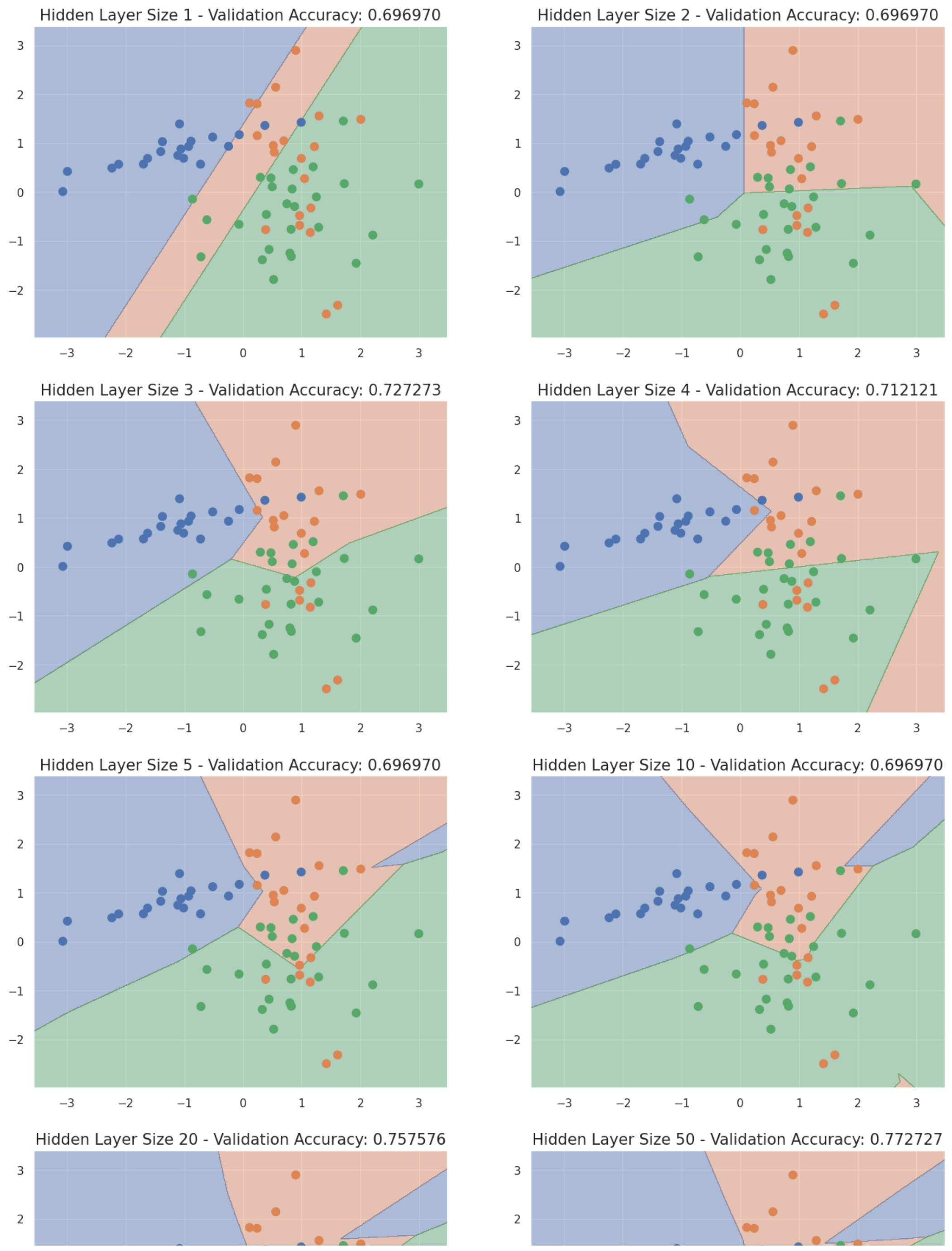
```
Loss after iteration 0: 5.136004
Loss after iteration 1000: 0.372306
Loss after iteration 2000: 0.335049
Loss after iteration 3000: 0.325840
Loss after iteration 4000: 0.320575
Loss after iteration 5000: 0.316680
Loss after iteration 6000: 0.313423
Loss after iteration 7000: 0.306506
```

Varying the Size of the Hidden Layer

With only three units in the hidden layer, the neural network is able to outperform the logistic classifier on the validation set. How does the size of the hidden layer impact the accuracy performance of the neural network?

```
Loss after iteration 14000: 0.280811
```

```
plt.figure(figsize=(16, 32)) # change default figure size
hidden_layer_dimensions = [1, 2, 3, 4, 5, 10, 20, 50]
for i, hidden_dim in enumerate(hidden_layer_dimensions):
    plt.subplot(5, 2, i+1)
    params, grads = train_nn(hidden_dim, num_passes=20000, dropout_ratio=None)
    nn_preds = forward(params, X_val, predict=True)
    plt.title('Hidden Layer Size %d - Validation Accuracy: %.6f' % (hidden_dim, np.mean(y_val==nn_preds)), fontsize=15)
    plot_decision_boundary(lambda x: forward(params, x, predict=True))
```



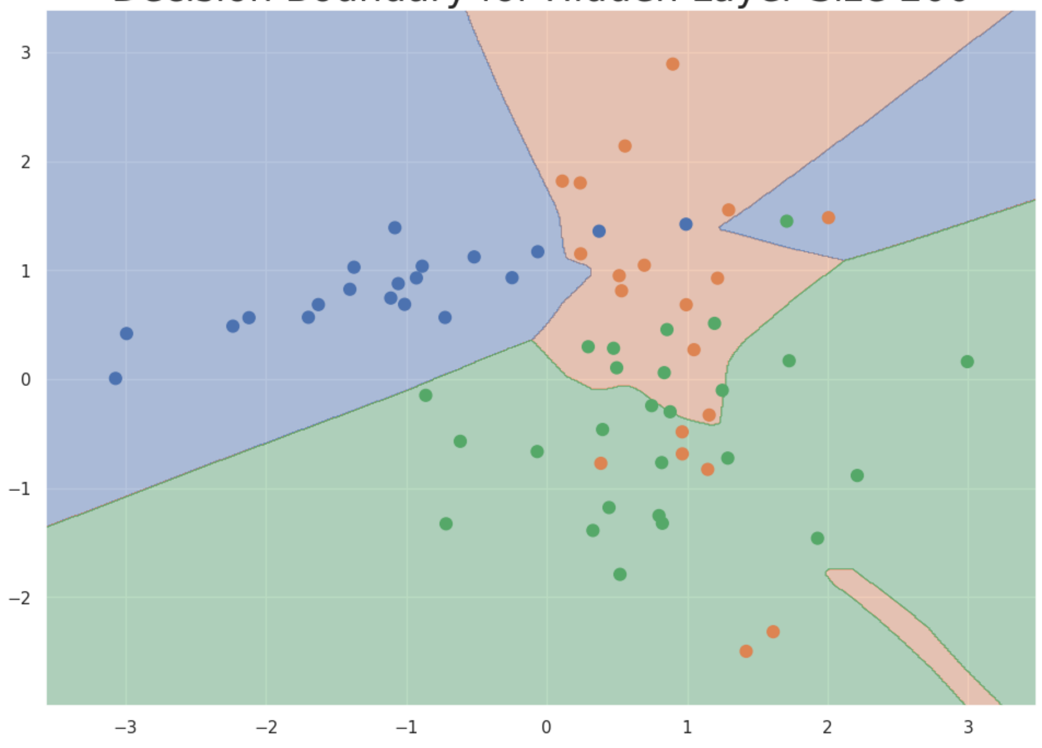
Overfitting

Let us try to overfit the neural network with a hidden layer of 200 units. The number of parameters the network needs to learn on a training set with two input features and three output classes is $W^{(1)} \in \mathbb{R}^{2\times 200}$, $b^{(1)} \in \mathbb{R}^{200}$, $W^{(2)} \in \mathbb{R}^{200\times 3}$, $b^{(2)} \in \mathbb{R}^{3}$, or $2 \times 200 + 200 + 200 \times 3 + 3 = 1,203$. How is accuracy performance affected by a network with 1,203 parameters relative to only 200 training examples?

```
# overfit the neural network by using 200 units in the hidden layer
hidden_dim = 200
params, _ = train_nn(hidden_dim, num_passes=20000, update_params=True, dropout_ratio=None, print_loss=1000)
# Plot the decision boundary
pred_func = lambda x: forward(params, x, predict=True)
plot_decision_boundary(pred_func)
plt.title('Decision Boundary for Hidden Layer Size {:d}'.format(hidden_dim), fontsize=fontsize)
nn_preds = forward(params, X_val, predict=True)

print('\nClassification Report for Hidden Layer Size {:d}:\n'.format(hidden_dim))
print(classification_report(y_val, nn_preds))
print('Validation Accuracy for Hidden Layer Size {:d}: {:.6f}'.format(hidden_dim, np.mean(y_val==nn_preds)))
```

```
Loss after iteration 0: 1.824944
Loss after iteration 1000: 0.384236
Loss after iteration 2000: 0.361828
Loss after iteration 3000: 0.348451
Loss after iteration 4000: 0.339086
Loss after iteration 5000: 0.331508
Loss after iteration 6000: 0.324729
Loss after iteration 7000: 0.318846
Loss after iteration 8000: 0.313890
Loss after iteration 9000: 0.309213
Loss after iteration 10000: 0.304782
Loss after iteration 11000: 0.300478
Loss after iteration 12000: 0.296362
Loss after iteration 13000: 0.292614
Loss after iteration 14000: 0.289069
Loss after iteration 15000: 0.285809
Loss after iteration 16000: 0.282678
Loss after iteration 17000: 0.279729
Loss after iteration 18000: 0.276910
Loss after iteration 19000: 0.274183

Classification Report for Hidden Layer Size 200:

              precision    recall  f1-score   support

           0       0.90      0.90      0.90        20
           1       0.57      0.65      0.60        20
           2       0.74      0.65      0.69        26

    accuracy                           0.73        66
   macro avg       0.73      0.73      0.73        66
weighted avg       0.74      0.73      0.73        66

Validation Accuracy for Hidden Layer Size 200: 0.727273
```

Decision Boundary for Hidden Layer Size 200



Dropout Regularization

We observe that while a hidden layer of low dimensionality nicely captures the general trend of the training data to produce good accuracy performance on the validation set, higher dimensionalities overfit on the training data and produce worse accuracy performance. In higher dimensionalities, the network is "memorizing" the data as opposed to learning the general patterns, and therefore does not generalize well on the validation set. Notice that we have already implemented weight decay regularization to help mitigate overfitting, but weight decay alone is just not enough in the case of limited training data.

Dropout is a simple yet effective network regularization technique that supplements L_2 weight decay. While training, dropout is implemented by only keeping some ratio of neurons (along with their connections) active with some probability, or setting them to zero otherwise. With dropout, the neural network is essentially different with each training pass, and thus has the property of preventing units from co-adapting too much. Dropout has been shown to improve the performance of neural networks on a wide variety of supervised learning tasks.

The code section below shows how dropout can overcome the adverse effects of overfitting to improve the accuracy performance of the network with a hidden layer of 200 units.

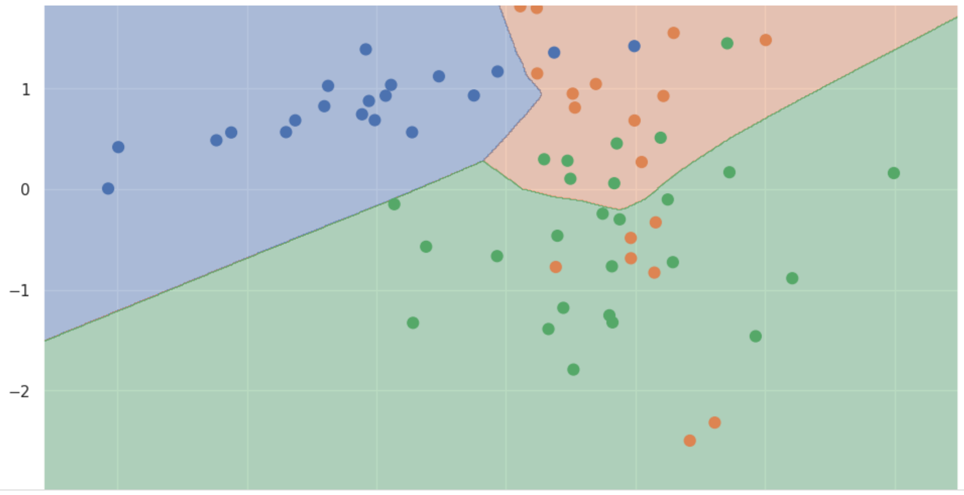
```
# fit neural network with 200 units in hidden layer, using dropout to mitigate overfitting
params, _ = train_nn(hidden_dim, num_passes=20000, update_params=True, dropout_ratio=0.5, print_loss=1000)
# Plot the decision boundary
pred_func = lambda x: forward(params, x, predict=True)
plot_decision_boundary(pred_func)
plt.title('Decision Boundary for Hidden Layer Size {:d} with Dropout'.format(hidden_dim), fontsize=fontsize)
nn_preds = forward(params, X_val, predict=True)

print('\nClassification Report with Dropout:\n')
print(classification_report(y_val, nn_preds))
print('Validation Accuracy with Dropout: {:.6f}'.format(np.mean(y_val==nn_preds)))
```

```
Loss after iteration 0: 2.057209
Loss after iteration 1000: 0.394945
Loss after iteration 2000: 0.378061
Loss after iteration 3000: 0.369220
Loss after iteration 4000: 0.361779
Loss after iteration 5000: 0.356664
Loss after iteration 6000: 0.352192
Loss after iteration 7000: 0.348074
Loss after iteration 8000: 0.345376
Loss after iteration 9000: 0.342109
Loss after iteration 10000: 0.339218
Loss after iteration 11000: 0.336260
Loss after iteration 12000: 0.334486
Loss after iteration 13000: 0.332938
Loss after iteration 14000: 0.329728
Loss after iteration 15000: 0.328432
Loss after iteration 16000: 0.325810
Loss after iteration 17000: 0.324478
Loss after iteration 18000: 0.322737
Loss after iteration 19000: 0.321375
```

Further Reading

The reader is encouraged to implement the additional tasks below to become more familiar with the code and further advance their understanding of neural network concepts:

1. Instead of batch gradient descent, use minibatch *stochastic* gradient descent to train the network. Minibatch stochastic gradient descent typically performs better in practice. (More information)
2. Extend our vanilla gradient descent implementation to include momentum update for better convergence on deep neural networks. (More information)
3. We used a fixed learning rate for gradient descent. Implement an annealing schedule for the learning rate during gradient descent updates. (More information)
4. We used the ReLU activation function for our hidden layer. Experiment with other activation functions (Sigmoid, Tanh, Leaky ReLU, Maxout). Note that changing the activation function also means changing the backpropagation derivatives. (More information)
5. Extend the network to include two hidden layers. Adding another hidden layer means you will need to adjust the code for both forward and backward propagation.



✓ 29s completed at 12:37 PM