Assignment 4 - Neural Networks

Version 1.2. Updated 2021.03.05

Zoe Zhu

Netid: yz656

Instructions for all assignments can be found here (https://github.com/kylebradbury/ids705/blob/master/assignments/_Assignment%20Instructions.ipynb), which is also linked to from the course syllabus (https://kylebradbury.github.io/ids705/index.html).

Learning objectives

Through completing this assignment you will be able to...

1. Identify key hyperparameters in neural networks and how they can impact model training and fit
2. Build, tune the parameters of, and apply feed-forward neural networks to data
3. Implement and explain each and every part of a standard fully-connected neural network and its operation including feed-forward propagation, backpropagation, and gradient descent.
4. Apply a standard neural network implementation and search the hyperparameter space for optimized application.
5. Develop a detailed understanding of the math and practical implementation considerations of neural networks, one of the most widely used machine learning tools.

In [1]:

```
import pandas as pd
import numpy as np
```

1

[50 points] Get to know your networks

The goal of this exercise is to better understand some of the key parameters used in neural networks so that you can be better prepared to tune your model. We'll be using the example data and data generation function below throughout this exercise.

In [2]:

```
#@title generate data
# Data generation function to create a checkerboard-patterned dataset
def make_data_checkerboard(n, noise=0):
    n_samples = int(n/4)
    scale = 5
    shift = 2.5
    center = 0.5
    c1a = (np.random.rand(n_samples,2)-center)*scale + [-shift, shift]
    c1b = (np.random.rand(n_samples,2)-center)*scale + [shift, -shift]
    c0a = (np.random.rand(n_samples,2)-center)*scale + [shift, shift]
    c0b = (np.random.rand(n_samples,2)-center)*scale + [-shift, -shift]
    X = np.concatenate((c1a,c1b,c0a,c0b),axis=0)
    y = np.concatenate((np.ones(2*n_samples), np.zeros(2*n_samples)))
    # Randomly flips a fraction of the labels to add noise
    for i,value in enumerate(y):
        if np.random.rand() < noise:
            y[i] = 1-value
    return (X,y)

# Training datasets (we create 3 to use to average over model)
np.random.seed(88)
N = 3
X_train = []
y_train = []
for i in range(N):
    Xt,yt = make_data_checkerboard(500, noise=0.25)
    X_train.append(Xt)
    y_train.append(yt)

# Validation and test data
X_val,y_val = make_data_checkerboard(3000, noise=0.25)
X_test,y_test = make_data_checkerboard(3000, noise=0.25)

# For the final performance evaluation, train on all of the training and validation dat
a:
X_train_plus_val = np.concatenate((X_train[0], X_train[1], X_train[2], X_val), axis=0)
y_train_plus_val = np.concatenate((y_train[0], y_train[1], y_train[2], y_val), axis=0)
```

The key parameters we want to explore the impact of are: learning rate, batch size, regularization coefficient, and the model architecture (number of layers and the number of nodes per layer). We'll explore each of these and determine an optimized configuration of the network for this problem. For all of the settings we'll explore, we'll assume the following default hyperparameters for the model (we'll use scikit learn's `MLPClassifier` (https://scikit-learn.org/stable/modules/generated/sklearn.neural_network.MLPClassifier.html#sklearn.neural_network.MLPC as our neural network model):

- `learning_rate_init` = 0.03
- `hidden_layer_sizes` = (30,30) (two hidden layers, each with 30 nodes)
- `alpha` = 0 (regularization penalty)
- `solver` = 'sgd' (stochastic gradient descent optimizer)
- `tol` = 1e-5 (this sets the convergence tolerance)
- `early_stopping` = False (this prevents early stopping)
- `activation` = 'relu' (rectified linear unit)
- `n_iter_no_change` = 1000 (this prevents early stopping)
- `batch_size` = 50 (size of the minibatch for stochastic gradient descent)
- `max_iter` = 500 (maximum number of epochs, which is how many times each data point will be used, not the number of gradient steps)

You'll notice we're eliminating early stopping so that we train the network the same amount for each setting. This allows us to compare the operation of the neural network while holding that value constant. Typically the amount of training would be another parameter to analyze the performance of.

(a) Visualize the impact of different hyperparameter settings. Starting with the default settings above make the following changes (only change one hyperparameter at a time). For each setting plot the decision boundary on the training data (since there are 3 training sets provided, use the first one to train on):

1. Vary the architecture (`hidden_layer_sizes`) by changing the number of nodes per layer while keeping the number of layers constant 2: (2,2), (5,5), (30,30)
2. Vary the learning rate: 0.0001, 0.01, 1
3. Vary the regularization: 0, 1, 10
4. Vary the batch size: 5, 50, 500

As you're exploring these settings, visit this website, the Neural Network Playground (https://playground.tensorflow.org/#activation=relu&batchSize=10&dataset=xor&regDataset=reg-plane&learningRate=0.03&regularizationRate=0&noise=20&networkShape=2,1&seed=0.89022&showTestData which will give you the chance to interactively explore the impact of each of these parameters not only on the mode output, but will also provide insight into a number of other important aspects of neural networks including: learning curves, batch size, and most importantly, the output of each intermediate neuron so that you can visualize the how neurons interact allowing you to combine them for more complex, nonlinear decision boundaries. As you're noting this, experiment by adding or removing hidden layers and neurons per layer. Vary the learning rate, regularization, and other settings.

In [3]:

```
from matplotlib import pyplot as plt
from matplotlib.colors import ListedColormap
from sklearn.neural_network import MLPClassifier
from mlxtend.plotting import plot_decision_regions

# Suppress warning
def warn(*args, **kwargs):
    pass
import warnings
warnings.warn = warn
```

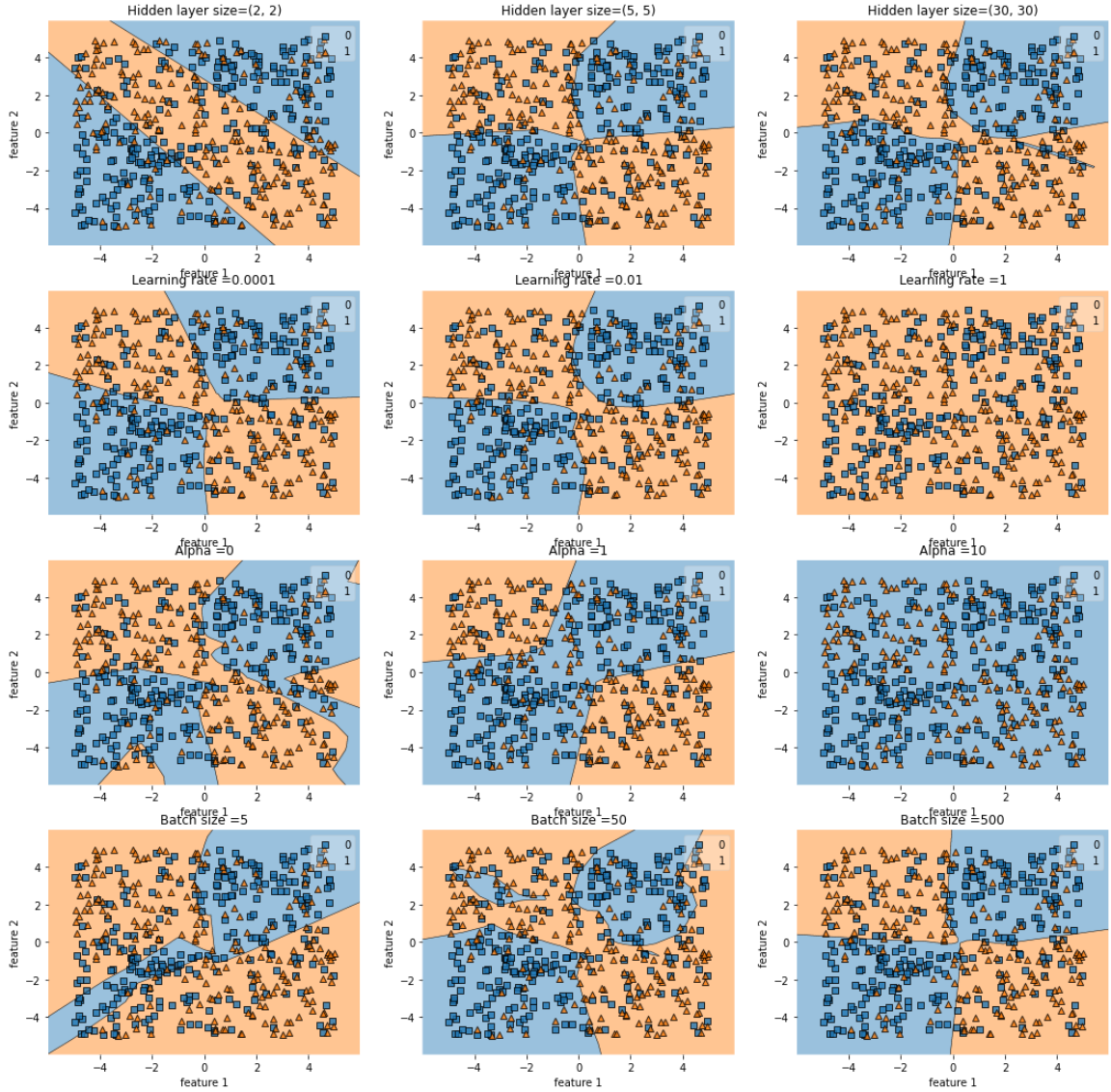
In [4]:

```
# Train on these following hyperparameters
hidden_layer_sizes_list = [(2,2),(5,5),(30,30)]
learning_rate_list = [0.0001, 0.01, 1]
regularization_list = [0, 1, 10]
batch_size_list = [5, 50, 500]
```

In [5]:

```
#@title 1(a)

# For each plot, plot the decision boundary on training data 1 - 4x3 plots?
# Set X = X_train[0], y = y_train[0]
plt.figure(figsize=(18, 18))

X = X_train[0]
y = y_train[0].astype(int)

i = 1
# Varying hidden layer size
for hidden_layer_sizes in hidden_layer_sizes_list:
  ax = plt.subplot(4, 3, i)
  clf = MLPClassifier(hidden_layer_sizes=hidden_layer_sizes,
                      learning_rate_init=0.03, alpha=0, batch_size=50,
                      solver = 'sgd',tol = 1e-5,early_stopping = False,activation = 're
lu',n_iter_no_change = 1000,max_iter = 500)
  clf.fit(X,y)

  fig = plot_decision_regions(X, y, clf)

  ax.set_xlabel('feature 1')
  ax.set_ylabel('feature 2')
  ax.set_title(f'Hidden layer size={hidden_layer_sizes}')

  i += 1


# Varying learning rate
for lr in learning_rate_list:
  ax = plt.subplot(4, 3, i)
  clf = MLPClassifier(learning_rate_init=lr,
                      hidden_layer_sizes=(30,30), alpha=0, batch_size=50,
                      solver = 'sgd',tol = 1e-5,early_stopping = False,activation = 're
lu',n_iter_no_change = 1000,max_iter = 500)
  clf.fit(X,y)

  fig = plot_decision_regions(X, y, clf)

  ax.set_xlabel('feature 1')
  ax.set_ylabel('feature 2')
  ax.set_title(f'Learning rate ={lr}')

  i += 1


# Varying regularization
for alpha in regularization_list:
  ax = plt.subplot(4, 3, i)
  clf = MLPClassifier(alpha=alpha,
                      learning_rate_init=0.03, hidden_layer_sizes=(30,30), batch_size=5
0,
                      solver = 'sgd',tol = 1e-5,early_stopping = False,activation = 're
lu',n_iter_no_change = 1000,max_iter = 500)
  clf.fit(X,y)

  fig = plot_decision_regions(X, y, clf)

  ax.set_xlabel('feature 1')
```

```
  ax.set_ylabel('feature 2')
  ax.set_title(f'Alpha ={alpha}')

  i += 1


# Varing batch size
for batch_size in batch_size_list:
  ax = plt.subplot(4, 3, i)
  clf = MLPClassifier(batch_size=batch_size,
                      learning_rate_init=0.03, hidden_layer_sizes=(30,30), alpha=0,
                      solver = 'sgd',tol = 1e-5,early_stopping = False,activation = 're
lu',n_iter_no_change = 1000,max_iter = 500)
  clf.fit(X,y)

  fig = plot_decision_regions(X, y, clf)

  ax.set_xlabel('feature 1')
  ax.set_ylabel('feature 2')
  ax.set_title(f'Batch size ={batch_size}')

  i += 1

plt.show()
```


Hidden layer size=(2, 2)
Hidden layer size=(5, 5)
Hidden layer size=(30, 30)
Learning rate =0.0001
Learning rate =0.01
Learning rate =1
Alpha =0
Alpha =1
Alpha =10
Batch size =5
Batch size =50
Batch size =500
feature 1
feature 2
0
1

(b) Now with some insight into which settings may work better than others, let's more fully explore the performance of these different settings in the context of our validation dataset. Holding all else constant (with the default settings mentioned above), vary each of the following parameters as specified below. Train your algorithm on the training data, and evaluate the performance of your algorithm on the validation dataset (here, overall accuracy is a reasonable performance metric since the classes are balanced and we don't weight one type of error as more important than the other); therefore, use the `score` method of the `MLPClassifier` for this. Create plot of accuracy vs each parameter you vary (this will be three plots).

1. Vary learning rate logarithmically from 10^{-5} to 10^{0} with 20 steps
2. Vary the regularization parameter logarithmically from 10^{-8} to 10^{2} with 20 steps
3. Vary the batch size over the following values: $[1, 3, 5, 10, 20, 50, 100, 250, 500]$

For each of these cases:

- Since neural networks can be sensitive to initialization values, run each of the settings above 3 times, and report the average accuracy in your plots (do not report the individual accuracy).
- Based on the results report your optimal choices for each of these hyperparameters and why you selected them.
- Use the chosen hyperparameter values as the new default settings for section (c) and (d).

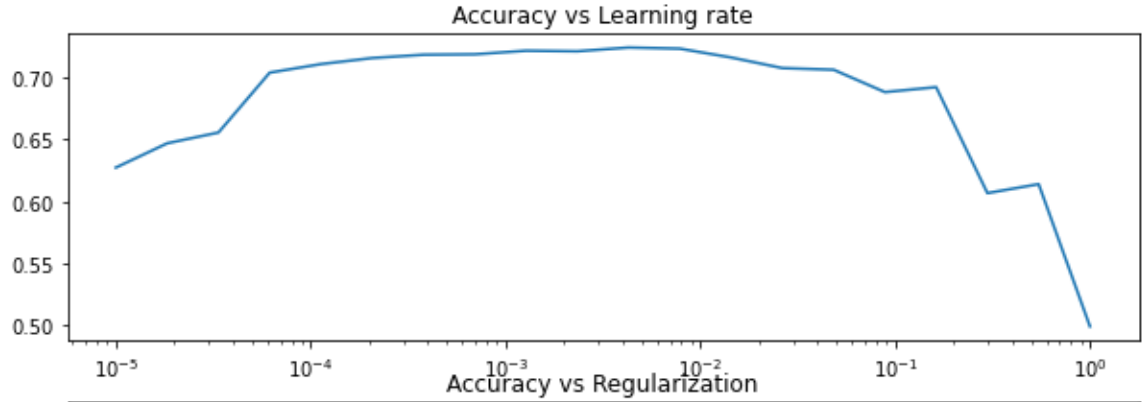
In [6]:

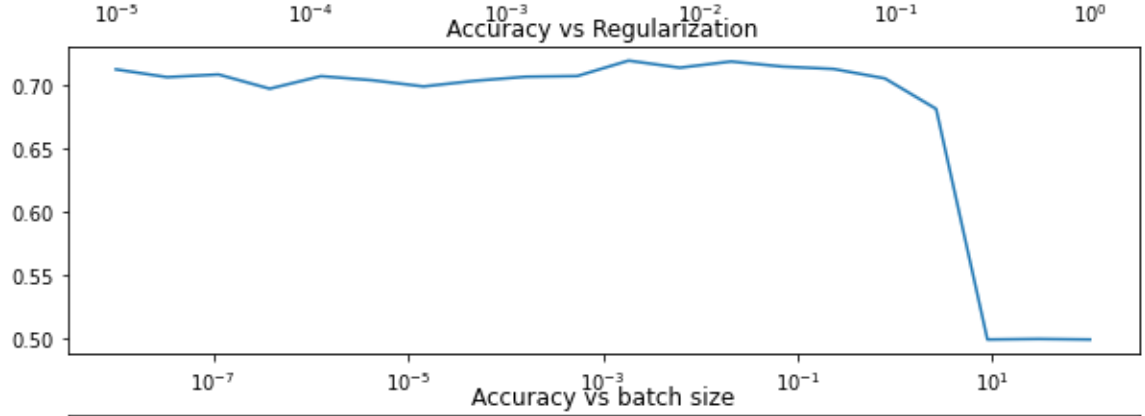
```
# Train on these following settings
learning_rate_list = np.logspace(-5,0, num=20)
regularization_list = np.logspace(-8,2,num=20)
batch_size_list = [1,3,5,10,20,50,100,250,500]
```

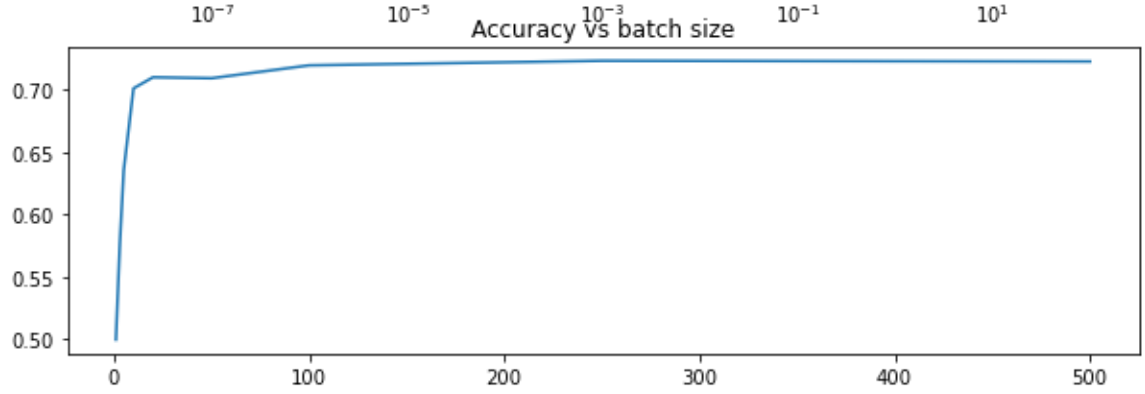
In [7]:

```
#@title 1(b)
warnings.warn = warn
plt.figure(figsize=(10, 10))

# Accuracy vs learning rate
plt.subplot(3,1,1)
mean_accuracy_list = []
for learning_rate in learning_rate_list:
  clf = MLPClassifier(learning_rate_init=learning_rate,
                      hidden_layer_sizes=(30,30),alpha=0,batch_size=50,
                      solver = 'sgd',tol = 1e-5,early_stopping = False,activation = 're
lu',n_iter_no_change = 1000,max_iter = 500)

  accuracy_list = []
  for i in range(0,3):
    clf.fit(X_train[i],y_train[i])
    score = clf.score(X_val,y_val)
    accuracy_list.append(score)

  mean_accuracy_list.append(np.mean(accuracy_list))

plt.plot(learning_rate_list,mean_accuracy_list)
plt.xscale('log')
plt.title('Accuracy vs Learning rate')


# Accuracy vs regularization
plt.subplot(3,1,2)
mean_accuracy_list = []
for regularization in regularization_list:
  clf = MLPClassifier(alpha=regularization,
                      hidden_layer_sizes=(30,30),batch_size=50, learning_rate_init=0.03
,
                      solver = 'sgd',tol = 1e-5,early_stopping = False,activation = 're
lu',n_iter_no_change = 1000,max_iter = 500)

  accuracy_list = []
  for i in range(0,3):
    clf.fit(X_train[i],y_train[i])
    score = clf.score(X_val,y_val)
    accuracy_list.append(score)

  mean_accuracy_list.append(np.mean(accuracy_list))

plt.plot(regularization_list,mean_accuracy_list)
plt.xscale('log')
plt.title('Accuracy vs Regularization')


# Accuracy vs batch size
plt.subplot(3,1,3)
mean_accuracy_list = []
for batch_size in batch_size_list:
  clf = MLPClassifier(batch_size=batch_size,
                      hidden_layer_sizes=(30,30),alpha=0, learning_rate_init=0.03,
                      solver = 'sgd',tol = 1e-5,early_stopping = False,activation = 're
lu',n_iter_no_change = 1000,max_iter = 500)

  accuracy_list = []
```

```
  for i in range(0,3):
    clf.fit(X_train[i],y_train[i])
    score = clf.score(X_val,y_val)
    accuracy_list.append(score)

  mean_accuracy_list.append(np.mean(accuracy_list))

plt.plot(batch_size_list,mean_accuracy_list)
plt.title('Accuracy vs batch size')


plt.show()
```


Accuracy vs Learning rate
0.70
0.65
0.60
0.55
0.50
10^{-5}
10^{-4}
10^{-3}
10^{-2}
10^{-1}
10^{0}


Accuracy vs Regularization
0.70
0.65
0.60
0.55
0.50
10^{-7}
10^{-5}
10^{-3}
10^{-1}
10^{1}


Accuracy vs batch size
0.70
0.65
0.60
0.55
0.50
0
100
200
300
400
500

ANSWER

Based on the results, I would choose the optimal learning rate to be 10^{-2}, the regularization to be 10^{-1} and the batch size to be 100 since the accuracy reaches the highest point and the trends are stable around the point selected.

In addition, these optimal values are chosen with consideration for computational efficiency. Looking at the results, there are actually a range of values with approximately the same performance in terms of accuracy. Specifically, there are no obvious changes in accuracy for learning rate from 10^{-4} to 10^{-2}, regularization from 10^{-3} to 10^{-1} and batch size after 100. Therefore, within the optimal range of accuracy, I would prefer hyperparameter values that speed up the training process.

However, these values are selected based on visual estimation and there is randomness in model training. These hyperparameter values are not necessarily the best choices for this model.

(c) Next we want to explore the impact of the model architecture but this means varying two parameters instead of one as above. To do this, evaluate the validation accuracy resulting from training the model using each pair of possible numbers of nodes per layer and number of layers from the lists below. We will assume that for any given configuration the number of nodes in each layer is the same (e.g. (2,2,2) and (25,25) are valid, but (2,5,3) is not). Use the optimized values for learning rate, regularization, and batch size selected from section (b).

- Number of nodes per layer: $[1, 2, 3, 4, 5, 10, 15, 25, 30]$
- Number of layers = $[1, 2, 3, 4]$ As in part (b), repeat this evaluation 3 times once for each of the three training sets, and report the average accuracy in your plots across those three trials (do not report the individual accuracy). For plotting these results use heatmaps to plot the data in two dimensions. To make the heatmaps, you can use [this code on creating heatmaps] https://matplotlib.org/stable/gallery/images_contours_and_fields/image_annotated_heatmap.html (https://matplotlib.org/stable/gallery/images_contours_and_fields/image_annotated_heatmap.html)). Be sure to include the numerical values of accuracy in each grid square as shown in the linked example and label your x, y, and color axes as always. For these numerical values, round them to **2 decimal places** (due to some randomness in the training process, any further precision is not typically meaningful).
- When you select your optimized parameters, be sure to keep in mind that these values may be sensitive to the data and may offer the potential to have high variance for larger models. Therefore, select the model with the highest accuracy but lowest number of total model weights.
- What do the results show? Which parameters did you select and why?

In [8]:

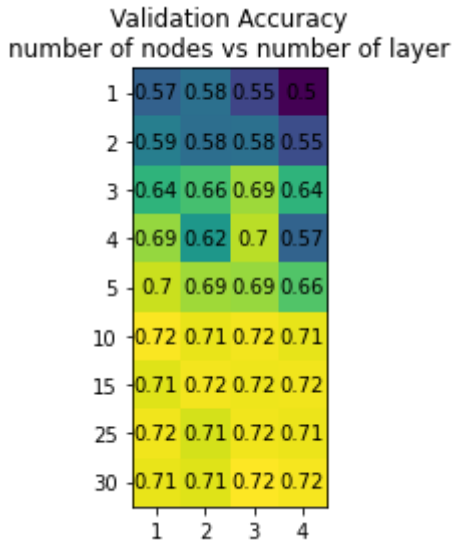
```
# From section b
best_lr = 0.001
best_alpha = 0.1
best_batch_size = 100

num_nodes = [1,2,3,4,5,10,15,25,30]
num_layers = [1,2,3,4]
```

In [9]:

```
mean_accuracy_list = []
for node in num_nodes:
  for layer in num_layers:
    clf = MLPClassifier(hidden_layer_sizes=tuple(np.repeat(node,layer)),
                        learning_rate_init=best_lr,
                        alpha=best_alpha,
                        batch_size=best_batch_size,
                        solver = 'sgd',tol = 1e-5,early_stopping = False,activation =
'relu',n_iter_no_change = 1000,max_iter = 500)
    accuracy_list = []
    for i in range(0,3):
      clf.fit(X_train[i],y_train[i])
      score = clf.score(X_val,y_val)
      accuracy_list.append(score)

    mean_accuracy_list.append(np.mean(accuracy_list))# len(num_nodes) x len(num_layer)
```

In [10]:

```
# plot Heatmap
acc = np.array(mean_accuracy_list).reshape(9,4)

fig, ax = plt.subplots()
im = ax.imshow(acc)

ax.set_xticks(np.arange(len(num_layers)))
ax.set_yticks(np.arange(len(num_nodes)))
ax.set_xticklabels(num_layers)
ax.set_yticklabels(num_nodes)

for i in range(len(num_nodes)):
    for j in range(len(num_layers)):
        text = ax.text(j, i, round(acc[i, j],2), ha="center", va="center", color="k")


ax.set_title("Validation Accuracy\nnumber of nodes vs number of layer")

fig.tight_layout()
plt.show()
```



The result shows that validation accuracy generally increases as we increase the number of nodes per layer and stops increasing after certain number of nodes. In this case, the validation accuracy changes little after 10 nodes per layer. However, there is no obvious influence of the number of layers on validation accuracy. When the number of nodes is comparatively small(1-5), the validation accuracy changes arbitrarily. When the number of nodes is comparatively large (>= 10), the validation accuracy are stable around 0.72.

To choose the most parsimonious model with best performance in terms of accuracy on validation set, I would select `hidden_layer_size` to be `(10,)` because the model with 1 layer and 10 nodes per layer achieved the highest validation accuracy with lowest number of total model weights.

(d) Based the optimal choice of hyperparameters and train your model with your optimized hyperparameters on all the training data (all three sets) AND the validation data (this is provided as `X_train_plus_val` and `y_train_plus_val`).

- Apply the trained model to the test data and report the accuracy of your final model on the test data.
- Plot an ROC curve of your performance (plot this with the curve in part (e) on the same set of axes if you complete that question).

In [11]:

```
best_node = 10
best_layer = 1

clf = MLPClassifier(hidden_layer_sizes=np.repeat(best_node,best_layer),
                    learning_rate_init=best_lr,
                    alpha=best_alpha,
                    batch_size=best_batch_size,
                    solver = 'sgd',tol = 1e-5,early_stopping = False,activation = 'rel
u',n_iter_no_change = 1000,max_iter = 500)
clf.fit(X_train_plus_val[:1500],y_train_plus_val[:1500])
score = clf.score(X_test,y_test)

print(f"Based the optimal choice of hyperparameters, the test accuracy is {score:.02f}"
)
```

```
Based the optimal choice of hyperparameters, the test accuracy is 0.70
```

(e) Automated hyperparameter search. The manual, greedy approach (setting one or two parameters at a time holding the rest constant), provides good insights into how the neural network hyperparameters impacts model fitting for this particular training process. However, it does limit our ability to more deeply search the hyperparameter space. Now we'll use a Scikit-Learn tool to search our hyperparameter space. Use `RandomizedSearchCV` (https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.RandomizedSearchCV.html#sklearn.model_selec to select the hyperparameters by training on ALL of the training and validation data (it will perform cross validation internally). You can use this example (https://scikit-learn.org/stable/auto_examples/model_selection/plot_randomized_search.html#sphx-glr-auto-examples-model-selection-plot-randomized-search-py) as a template for how to do this. Grid search searches all possible combinations of values entered as possible values. Doing this over a large hyperparameter space for a model that takes awhile to run is intractable. Random search has been shown to be surprisingly effective in these situations at identifying excellent

- Set the number of iterations to at least 25 (you'll look at 25 random pairings of possible parameters). You can go as high as you want, but it will take longer the larger this value is.
- If you run this on Colab or any system with multiple cores, set the parameter `n_jobs` to -1 to use all available cores for more efficient training through parallelization
- You'll need to set the range or distribution of the parameters you want to sample from. Search over the same ranges as in previous problems (except this time, you'll search over all the parameters at once). You can use lists of values for batch_size, `loguniform` for the learning rate and regularization parameter, and a list of tuples for the `hidden_layer_sizes` parameter.
- Once the model is fit, use the `best_params_` attribute to extract the optimized values of the parameters
- State the accuracy of the model on the test dataset
- Plot the ROC curve corresponding to your best model through greedy hyperparameter section vs the model identified through random search. In the legend of the plot, report the AUC for each curve
- Plot the final decision boundary for the greedy and random search-based classifiers along with one of the training datasets to demonstrate the shape of the final boundary
- How did the performance compare?

In [12]:

```
from time import time
import scipy.stats as stats
from sklearn.utils.fixes import loguniform
from sklearn.model_selection import RandomizedSearchCV

clf = MLPClassifier(solver = 'sgd',tol = 1e-5,early_stopping = False,activation = 'rel
u',n_iter_no_change = 1000,max_iter = 500)

# Generate a list for hidden_layer_size
hidden_layer_list = []

for node in num_nodes:
  for layer in num_layers:
    hidden_layer_list.append(tuple(np.repeat(node,layer)))
```

In [13]:

```
def report(results, n_top=1):
    for i in range(1, n_top + 1):
        candidates = np.flatnonzero(results['rank_test_score'] == i)
        for candidate in candidates:
            print("Model with rank: {0}".format(i))
            print("Mean validation score: {0:.2f} (std: {1:.2f})"
                  .format(results['mean_test_score'][candidate],
                          results['std_test_score'][candidate]))
            print("Parameters: {0}".format(results['params'][candidate]))
            print("")
```

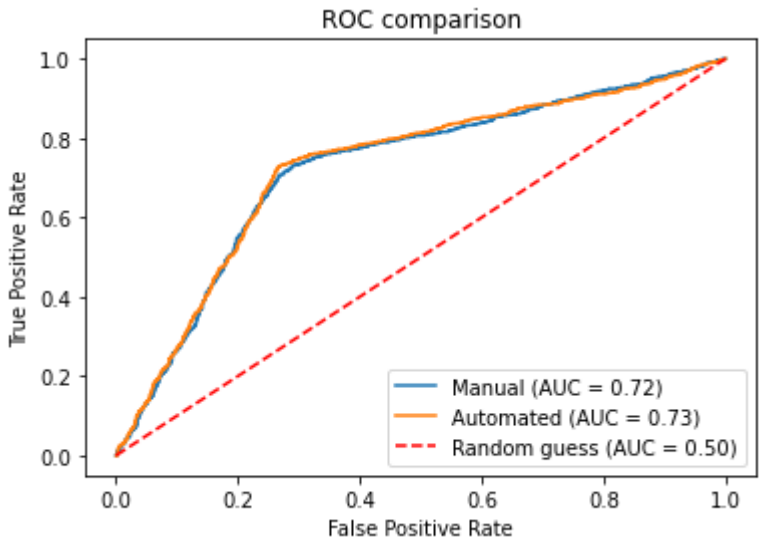
In [14]:

```
# specify parameters and distributions to sample from

param_dist = {'hidden_layer_sizes': hidden_layer_list,
              'learning_rate_init':loguniform(1e-5, 1e0),
              'batch_size': np.arange(25,500,5),
              'alpha':loguniform(1e-5, 1e0) }

# run randomized search, at least 25
n_iter_search = 25
random_search = RandomizedSearchCV(clf, param_distributions=param_dist,n_iter=n_iter_se
arch,n_jobs=-1)
start = time()
random_search.fit(X_train_plus_val, y_train_plus_val)
print("RandomizedSearchCV took %.2f seconds for %d candidates"
      " parameter settings." % ((time() - start), n_iter_search))
report(random_search.cv_results_)
```

```
RandomizedSearchCV took 453.32 seconds for 25 candidates parameter setting
s.
Model with rank: 1
Mean validation score: 0.73 (std: 0.13)
Parameters: {'alpha': 0.0017003636103905303, 'batch_size': 155, 'hidden_la
yer_sizes': (10, 10, 10), 'learning_rate_init': 0.006590429647721945}
```

In [15]:

```
from sklearn.metrics import roc_curve, auc

rs_alpha = random_search.best_params_['alpha']
rs_batch_size = random_search.best_params_['batch_size']
rs_hidden = random_search.best_params_['hidden_layer_sizes']
rs_lr = random_search.best_params_['learning_rate_init']

# manual,greedy approach
clf_gs = MLPClassifier(hidden_layer_sizes=np.repeat(best_node,best_layer),
                       learning_rate_init=best_lr,
                       alpha=best_alpha,
                       batch_size=best_batch_size,
                       solver = 'sgd',tol = 1e-5,early_stopping = False,activation = 'r
elu',n_iter_no_change = 1000,max_iter = 500)
y_pred = clf_gs.fit(X_train_plus_val[:1500],y_train_plus_val[:1500]).predict_proba(X_te
st)[:, 1]
fpr_gs, tpr_gs, _ = roc_curve(y_test, y_pred)

# random search
clf_rs = MLPClassifier(hidden_layer_sizes=rs_hidden,
                       learning_rate_init=rs_lr,
                       alpha=rs_alpha,
                       batch_size=rs_batch_size,
                       solver = 'sgd',tol = 1e-5,early_stopping = False,activation = 'r
elu',n_iter_no_change = 1000,max_iter = 500)
y_pred = clf_rs.fit(X_train_plus_val[:1500],y_train_plus_val[:1500]).predict_proba(X_te
st)[:, 1]
fpr_rs, tpr_rs, _ = roc_curve(y_test, y_pred)

plt.plot(fpr_gs, tpr_gs, label=f"Manual (AUC = {auc(fpr_gs, tpr_gs):.02f})")
plt.plot(fpr_rs, tpr_rs, label=f"Automated (AUC = {auc(fpr_rs, tpr_rs):.02f})")
plt.plot([0, 1], [0, 1], color="red", linestyle="--", label="Random guess (AUC = 0.50)"
)
plt.xlabel("False Positive Rate")
plt.ylabel("True Positive Rate")
plt.title("ROC comparison")
plt.legend()
plt.show()
```



In [16]:

```
# Decision boundary comparison
plt.figure(figsize=(15, 5))
# manual
ax = plt.subplot(1,2,1)
fig = plot_decision_regions(X, y, clf_gs)

ax.set_xlabel('feature 1')
ax.set_ylabel('feature 2')
ax.set_title(f'Manual hyperparameter search on training set 1')

# random search
ax = plt.subplot(1,2,2)
fig = plot_decision_regions(X, y, clf_rs)

ax.set_xlabel('feature 1')
ax.set_ylabel('feature 2')
ax.set_title(f'Automated hyperparameter search on training set 1')

plt.show()
```

ANSWER

In [19]:

```
print(f"Based the optimal choice of hyperparameters from manual selection, the test acc
uracy is {clf_gs.score(X_test,y_test):.02f}")
print(f"Based the optimal choice of hyperparameters from random search, the test accura
cy is {clf_rs.score(X_test,y_test):.02f}")
```

```
Based the optimal choice of hyperparameters from manual selection, the tes
t accuracy is 0.71
Based the optimal choice of hyperparameters from random search, the test a
ccuracy is 0.73
```

Comparing the test accuracy, the model with optimal hyperparameter values performs better than the model with manually selected values. ROC curves seem similar for two models while automated hyperparameter search yields a slightly higher AUC. For the final decision boundary, the model with optimal hyperparameter values has a decision boundary approximates to the checkerboard, which corresponds to the pattern in the data simulation. Therefore, automated hyperparameter search outperforms manual selection on this task.