Visualizing aspects of NN training

We do 3 types of simple visualizations surrounding the training of our NNs.

1. Data visualization

In this step, we're going to examine our training data.

It's important to examine your data before you start training your models on it, so that you have a good idea of what kind of data your working on, and what kinds of errors you should look out for.

2. Training Visualization

In the second step, we're going to visualize the training and validation losses of our models as we train them.

This is a very important aspect of training deep models. Looking at a graph of these 2 losses tells us a lot about how well our model is learning what we want it to.

3. Hyperparameter Visualization

The last step of visualization is hyperparameter search.

The best we can do is run the model multiple times on different values, and pick the ones that do the best on our development data.

We take turns fixing all of our hyperparameters except one, and examining how our performance changes as we change that one parameter.

1. Visualizing our data

In [1]:
```
from collections import defaultdict
import numpy as np
from data import load
import matplotlib.pyplot as plt
%matplotlib inline
```

In [2]:
```
# Load the data

data, labels = load('./release-data', split='train')
dev_data, dev_labels = load('./release-data', split='dev')
```

1.1 The label distribution

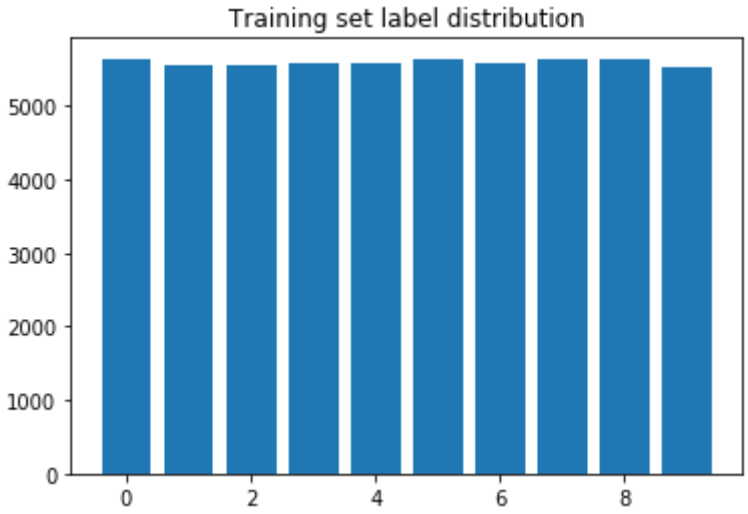
The first thing we'll examine is our training set label distribution. We'd like to know what our training set looks like, and how it compares to the distribution of our validation set.

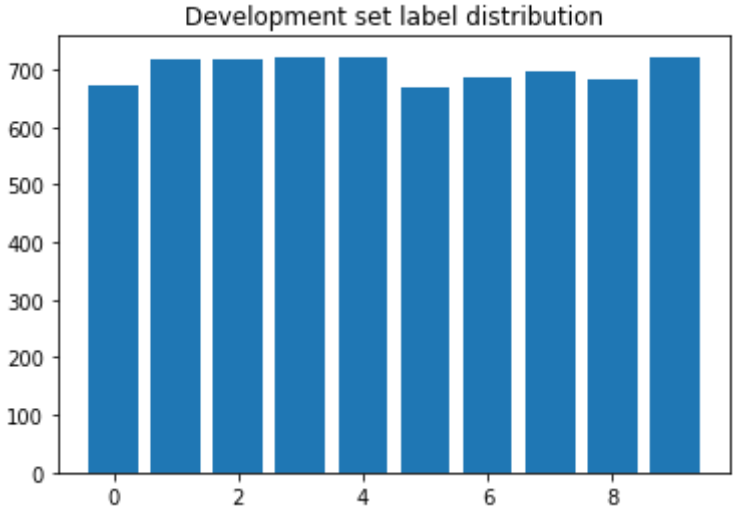
If things don't look very similar, we might have a *domain mismatch*. Adapting models trained from a different domain that what they're being evaluated on is a very active area of research, currently.

In [3]:
```
counts = defaultdict(int)
for label in labels:
    counts[label] += 1
plt.figure()
plt.title("Training set label distribution")
k = counts.keys()
v = counts.values()
plt.bar(list(k), height=list(v))

dev_counts = defaultdict(int)
for label in dev_labels:
    dev_counts[label] += 1
plt.figure()
plt.title("Development set label distribution")
dk = dev_counts.keys()
dv = dev_counts.values()
plt.bar(list(dk), height=list(dv))
```

Out[3]: <BarContainer object of 10 artists>





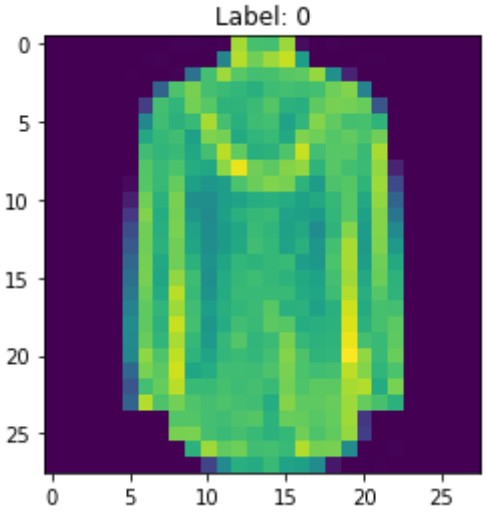
Luckily, our label distribution looks fairly even across train and dev, so we won't worry about domain adaptation techniques. We'll just train our models assuming that the training and test distributions are equal.

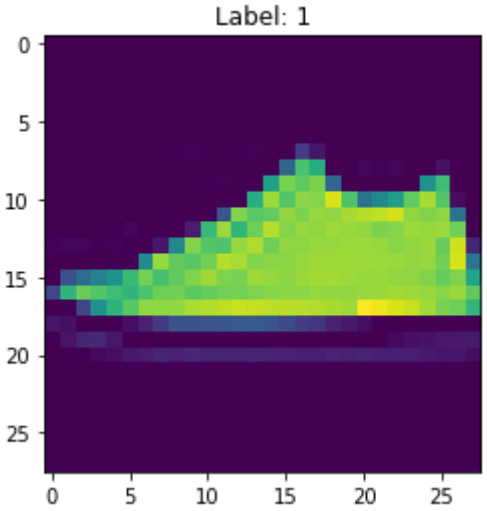
1.2 Visualizing our data

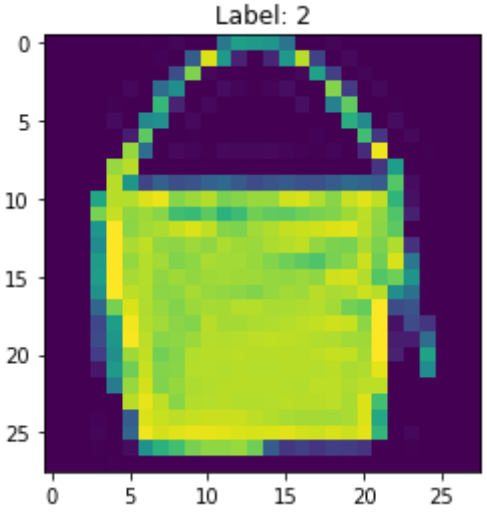
Next, let's visualize some of our data points. Since the dataset we're working on is a vision dataset, we can actually look at our data as images.

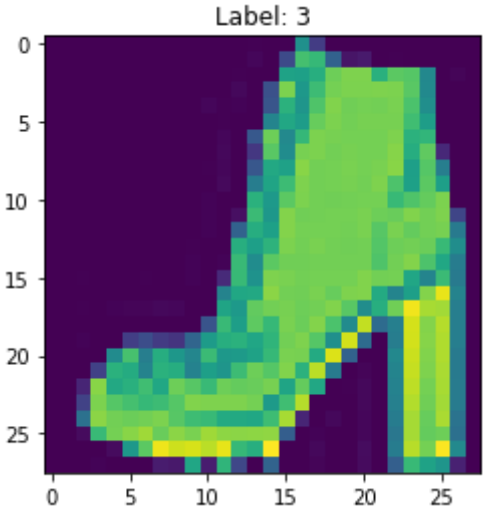
Let's take a look at one example for each label that we have.

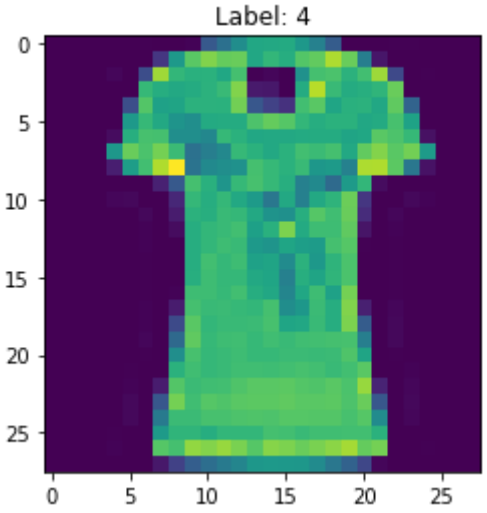
In [4]:
```
# Plot some examples of the data
for label in range(10):
    for i in range(len(labels)):
        if int(labels[i]) == label:
            label_idx = i
            break
    plt.figure()
    plt.title(f"Label: {label}")
    ex = np.array(data[label_idx], dtype=float)
    plt.imshow(ex.reshape((28,28)))
```

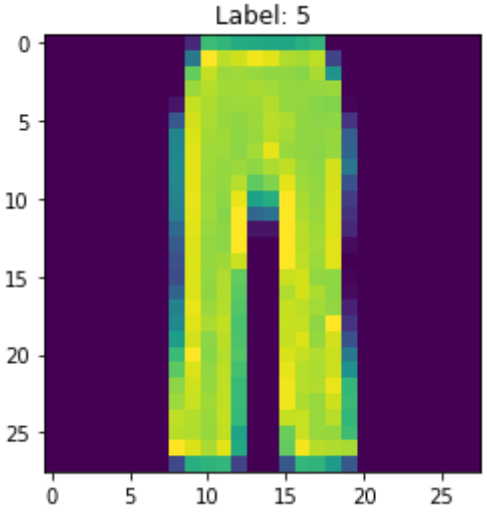

Label: 0
0
5
10
15
20
25
0
5
10
15
20
25

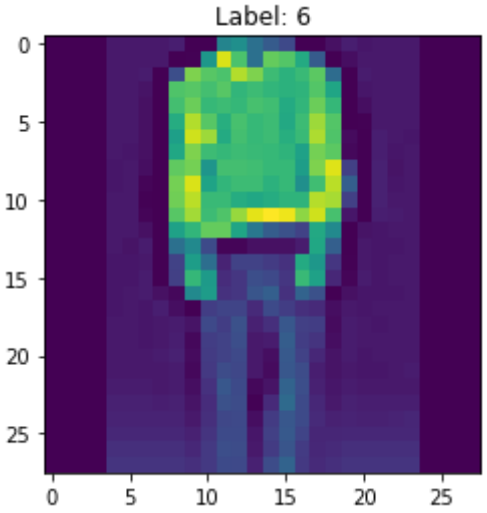

Label: 1
0
5
10
15
20
25
0
5
10
15
20
25

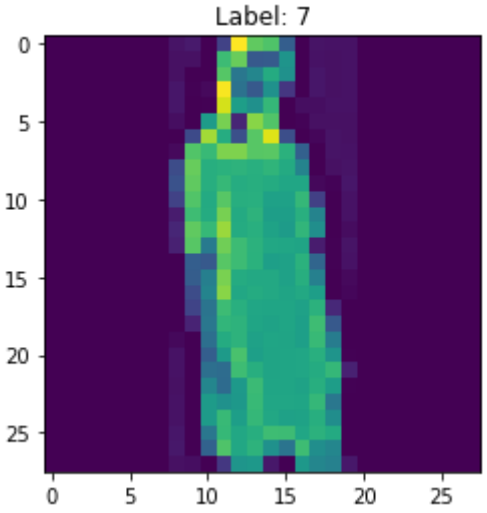

Label: 2
0
5
10
15
20
25
0
5
10
15
20
25

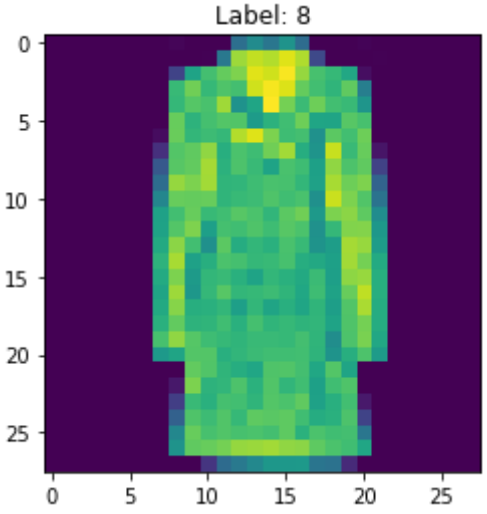

Label: 3
0
5
10
15
20
25
0
5
10
15
20
25

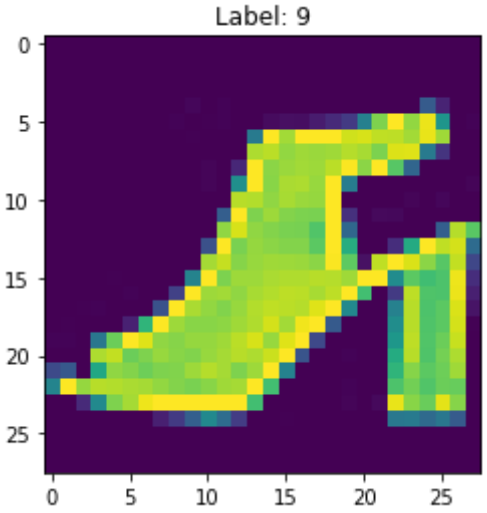

Label: 4
0
5
10
15
20
25
0
5
10
15
20
25


Label: 5
0
5
10
15
20
25
0
5
10
15
20
25


Label: 6
0
5
10
15
20
25
0
5
10
15
20
25


Label: 7
0
5
10
15
20
25
0
5
10
15
20
25


Label: 8
0
5
10
15
20
25
0
5
10
15
20
25



2. Visualizing our model's learning

An important aspect of training DNN models is visualizing the training and development loss as our models train, as well as the accuracies.

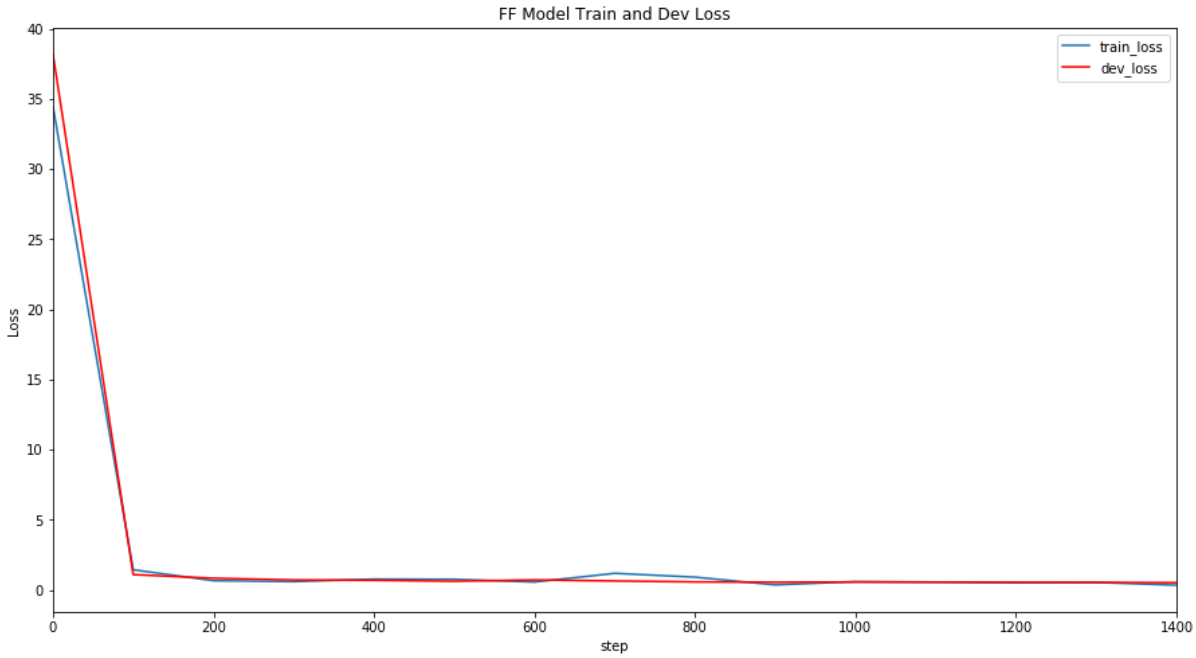
These graphs can tell us a lot about how well our models are doing. For instance, if we see that our training loss is going down, but our dev loss starts going up, we know that we are overfitting and we should have stopped training.

In [5]:
```
import pandas as pd
import matplotlib.pyplot as plt
%matplotlib inline
```

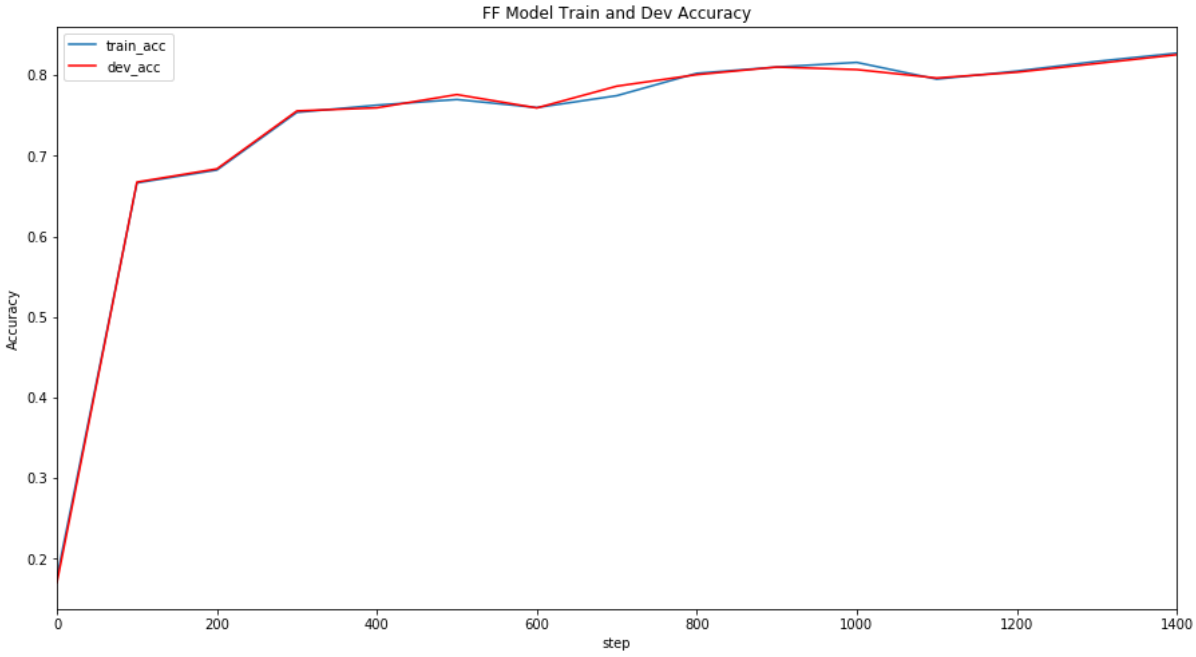
2.1 Feed Forward Model

In [6]:
```
ff_metrics = pd.read_csv('best-ff-logs.csv')
```

In [7]:
```
plt.figure(figsize=(15, 8))
plt.title("FF Model Train and Dev Loss")
ax = plt.gca()
plt.ylabel('Loss')
ff_metrics.plot(kind='line',x='step',y='train_loss',ax=ax)
ff_metrics.plot(kind='line',x='step',y='dev_loss', color='red', ax=ax)
plt.show()
```



In [8]:
```
plt.figure(figsize=(15, 8))
plt.title("FF Model Train and Dev Accuracy")
ax = plt.gca()
plt.ylabel('Accuracy')
ff_metrics.plot(kind='line',x='step',y='train_acc',ax=ax)
ff_metrics.plot(kind='line',x='step',y='dev_acc', color='red', ax=ax)
plt.show()
```



Let's look at the accuracy graph to check if we need to train our models for more steps.

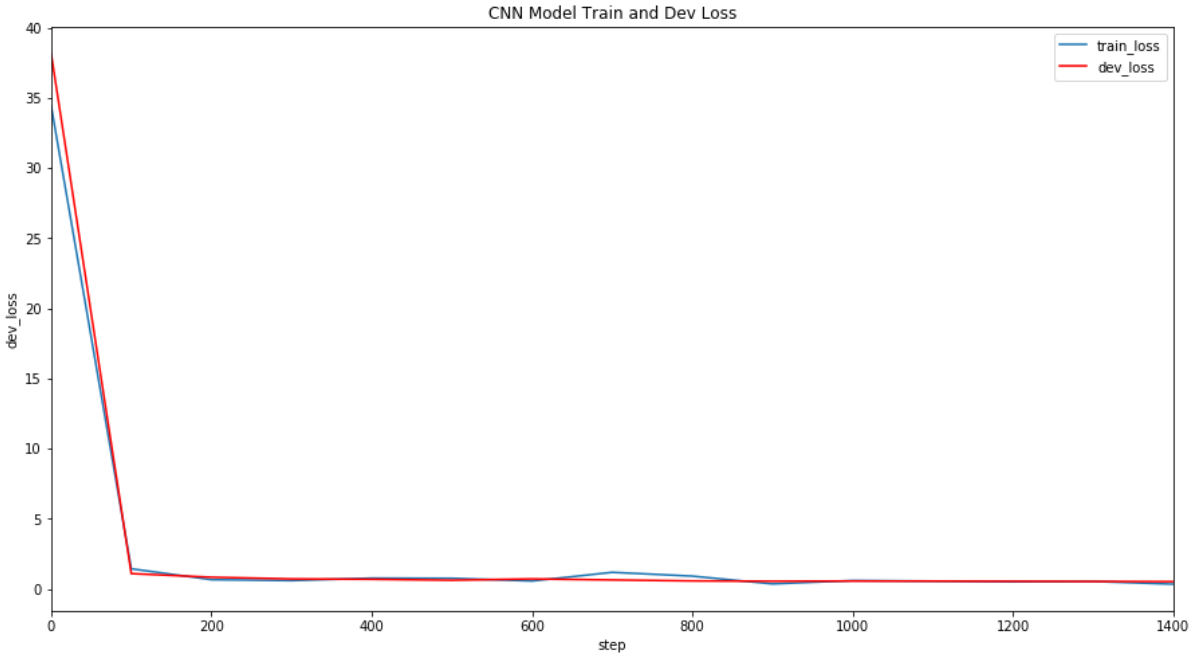
The graph shows that dev accuracy is still trending upwards. So it is likely that training for more steps will increase it. Training should be stopped when dev accuracy plateaus.

2.2 Basic CNN Model

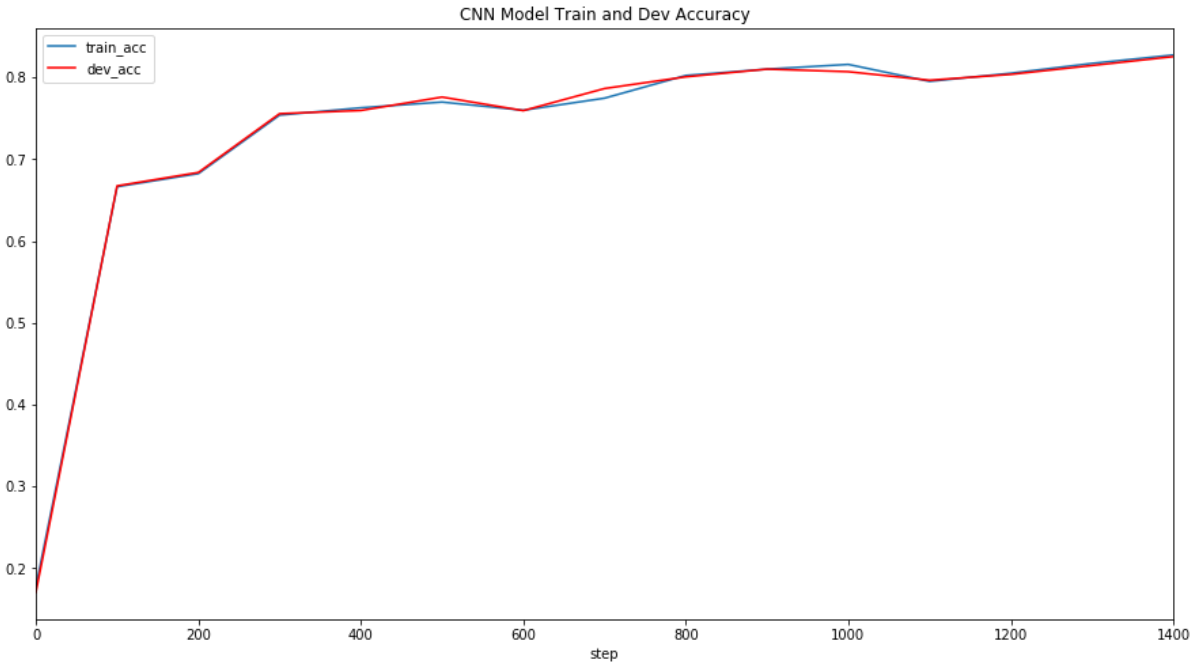
This graph is quite a bit smoother than the FF model. This model takes many more steps to train than the FF model!

In [9]:
```
cnn_metrics = pd.read_csv('best-cnn-logs.csv') # point this to the corre
ct log file!
```

In [10]:
```
plt.figure(figsize=(15, 8))
plt.title("CNN Model Train and Dev Loss")
ax = plt.gca()
plt.ylabel('dev_loss')
ff_metrics.plot(kind='line',x='step',y='train_loss',ax=ax)
ff_metrics.plot(kind='line',x='step',y='dev_loss', color='red', ax=ax)
plt.show()
```



In [11]:
```
plt.figure(figsize=(15, 8))
plt.title("CNN Model Train and Dev Accuracy")
ax = plt.gca()
ff_metrics.plot(kind='line',x='step',y='train_acc',ax=ax)
ff_metrics.plot(kind='line',x='step',y='dev_acc', color='red', ax=ax)
plt.show()
```

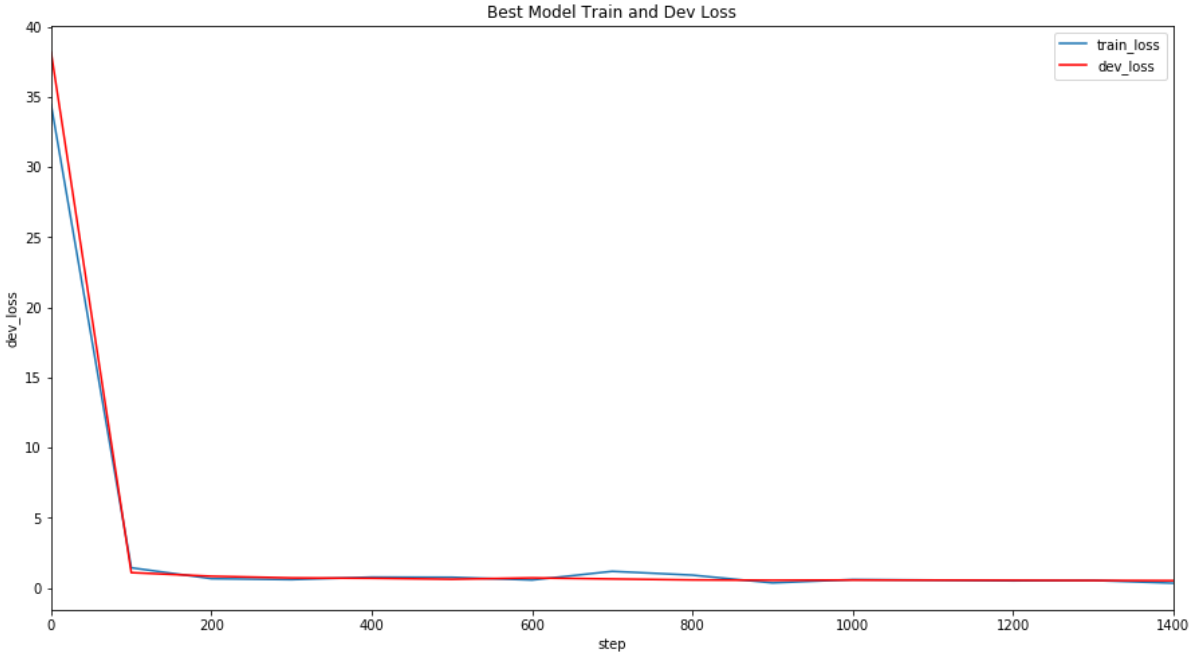


By the looks of it, the model has not converged. Increasing the number of steps even more would be the first thing to do. Training is slightly unstable. Adding batch normalization should help. It might also indicate that a slightly lower learning rate would be better.

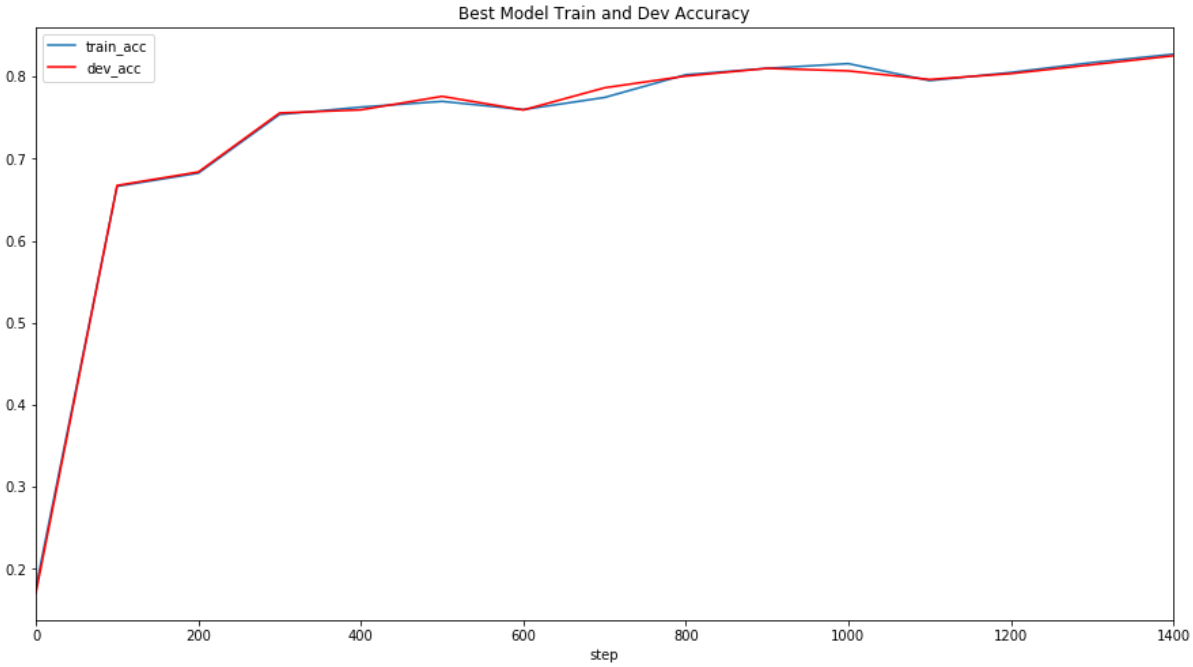
2.3 Best Model

In [12]:
```
best_metrics = pd.read_csv('best-hybrid-logs.csv')
```

In [13]:
```
plt.figure(figsize=(15, 8))
plt.title("Best Model Train and Dev Loss")
ax = plt.gca()
plt.ylabel('dev_loss')
ff_metrics.plot(kind='line',x='step',y='train_loss',ax=ax)
ff_metrics.plot(kind='line',x='step',y='dev_loss', color='red', ax=ax)
plt.show()
```



In [14]:
```
plt.figure(figsize=(15, 8))
plt.title("Best Model Train and Dev Accuracy")
ax = plt.gca()
ff_metrics.plot(kind='line',x='step',y='train_acc',ax=ax)
ff_metrics.plot(kind='line',x='step',y='dev_acc', color='red', ax=ax)
plt.show()
```



3. Visualizing Hyperparameter Search

Complex models have tons of hyperparameters!

We're going to be performing a hyperparameter sweep over certain hyperparameters.

To do this, we freeze all hyperparameters except one. We will then choose a range of values for the one unfrozen hyperparameter. We'll train an individual model for each value of the range, and then store it's performance on held out development data.

Once we have dev accuracy for each setting, we can plot it and examine the learning trends as we sweep over that hyperparameters.

Ideally, if we have chosen an effective range, we should see something of an upside-down "U" shape, indicating that we have pushed the hyperparameter to either extreme where it starts to hurt performance, and we might have found something of an optimal value.

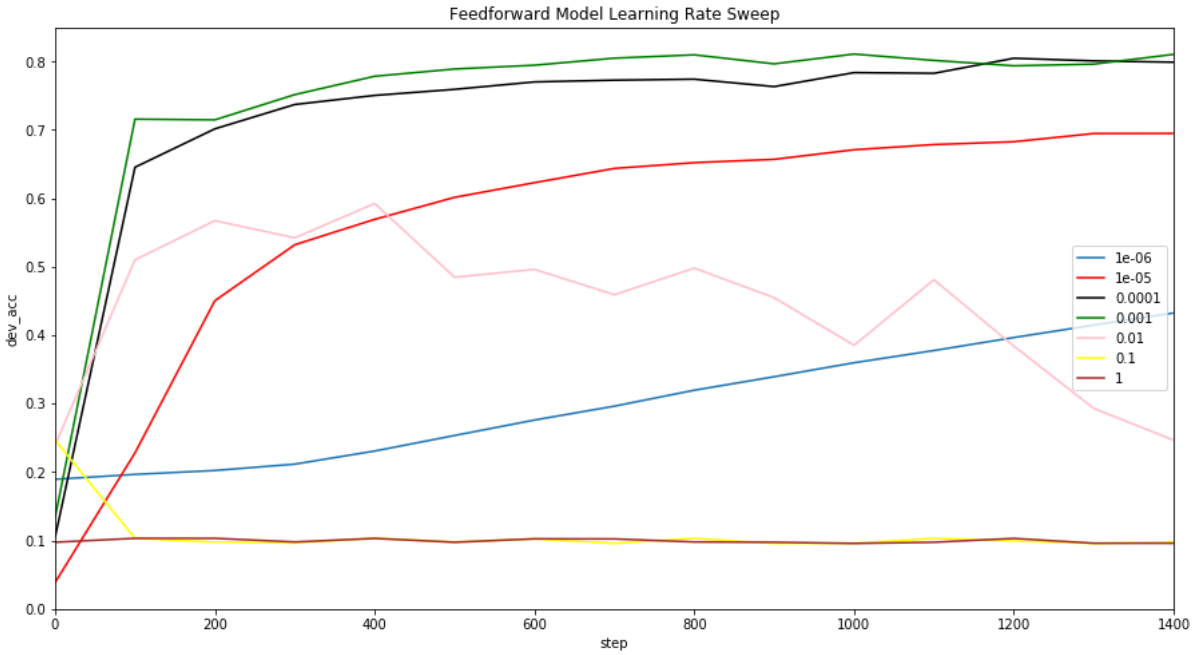
Of course, this upside-down "U" and it's maximum value might only be true in the setting where all of our other hyperparameters are frozen. If we change those, then the sweep we just did might not be correct anymore. However, if we've chosen good values to freeze our other hyperparameters with, then we should learn something about the hyperparameter we're examining.

3.1 Feed Forward Learning Rate Sweep

Let's examine the effect of increasing and decreasing the initial learning rate.

In [15]:
```
ff_metrics_0 = pd.read_csv('ff0-logs.csv')
ff_metrics_1 = pd.read_csv('ff1-logs.csv')
ff_metrics_2 = pd.read_csv('ff2-logs.csv')
ff_metrics_3 = pd.read_csv('ff3-logs.csv')
ff_metrics_4 = pd.read_csv('ff4-logs.csv')
ff_metrics_5 = pd.read_csv('ff5-logs.csv')
ff_metrics_6 = pd.read_csv('ff6-logs.csv')
```

In [16]:
```
plt.figure(figsize=(15, 8))
plt.title("Feedforward Model Learning Rate Sweep")
ax = plt.gca()
plt.ylabel('dev_acc')
ff_metrics_0.plot(kind='line',x='step',y='dev_acc',ax=ax,label=0.000001)
ff_metrics_1.plot(kind='line',x='step',y='dev_acc',ax=ax,label=0.00001,c
olor='red')
ff_metrics_2.plot(kind='line',x='step',y='dev_acc',ax=ax,label=0.0001,co
lor='black')
ff_metrics_3.plot(kind='line',x='step',y='dev_acc',ax=ax,label=0.001,col
or='green')
ff_metrics_4.plot(kind='line',x='step',y='dev_acc',ax=ax,label=0.01,colo
r='pink')
ff_metrics_5.plot(kind='line',x='step',y='dev_acc',ax=ax,label=0.1,color
='yellow')
ff_metrics_6.plot(kind='line',x='step',y='dev_acc',ax=ax,label=1,color=
'brown')
plt.show()
```



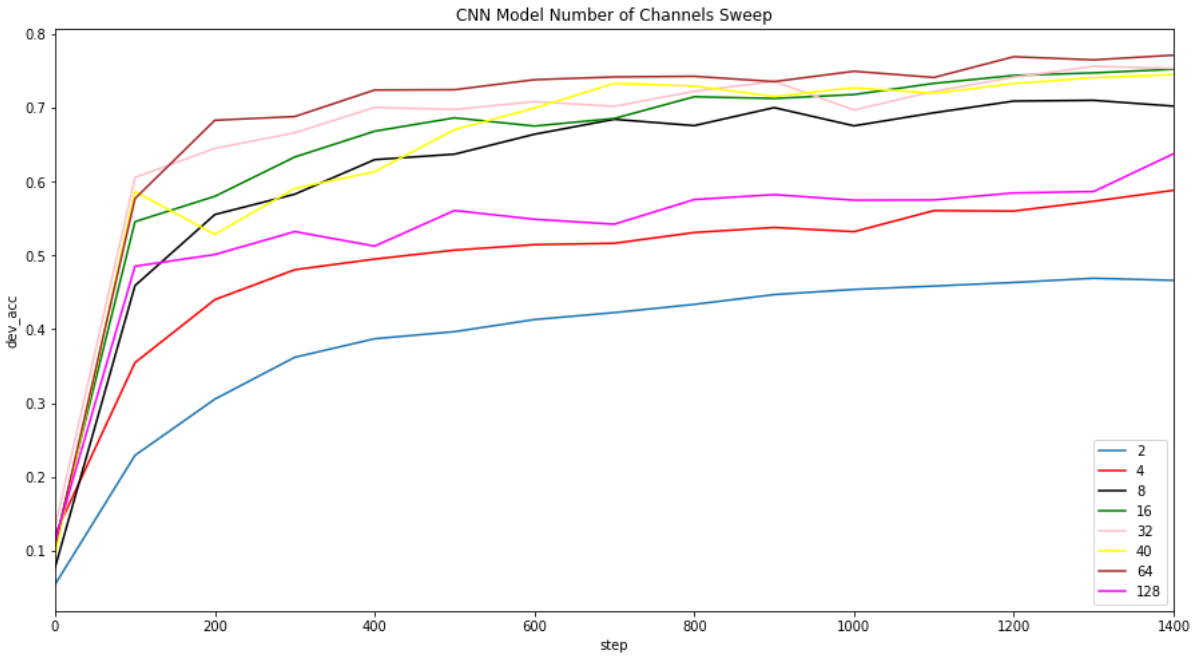
This plot shows that having too high a learning rate makes training unstable, while having a really low learning rate makes it incredibly slow to train. Here, the ideal learning rate is 0.001.

3.2 CNN Number of Channels Sweep

We take a look at how choice of channel size for conv1 affects the CNN model we've implemented.

In [17]:
```
cnn_metrics_0 = pd.read_csv('cnn0-logs.csv')
cnn_metrics_1 = pd.read_csv('cnn1-logs.csv')
cnn_metrics_2 = pd.read_csv('cnn2-logs.csv')
cnn_metrics_3 = pd.read_csv('cnn3-logs.csv')
cnn_metrics_4 = pd.read_csv('cnn4-logs.csv')
cnn_metrics_5 = pd.read_csv('cnn5-logs.csv')
cnn_metrics_6 = pd.read_csv('cnn6-logs.csv')
cnn_metrics_7 = pd.read_csv('cnn7-logs.csv')
```

In [18]:
```
plt.figure(figsize=(15, 8))
plt.title("CNN Model Number of Channels Sweep")
ax = plt.gca()
plt.ylabel('dev_acc')
cnn_metrics_0.plot(kind='line',x='step',y='dev_acc',ax=ax,label=2)
cnn_metrics_1.plot(kind='line',x='step',y='dev_acc',ax=ax,label=4,color=
'red')
cnn_metrics_2.plot(kind='line',x='step',y='dev_acc',ax=ax,label=8,color=
'black')
cnn_metrics_3.plot(kind='line',x='step',y='dev_acc',ax=ax,label=16,color
='green')
cnn_metrics_4.plot(kind='line',x='step',y='dev_acc',ax=ax,label=32,color
='pink')
cnn_metrics_5.plot(kind='line',x='step',y='dev_acc',ax=ax,label=40,color
='yellow')
cnn_metrics_6.plot(kind='line',x='step',y='dev_acc',ax=ax,label=64,color
='brown')
cnn_metrics_7.plot(kind='line',x='step',y='dev_acc',ax=ax,label=128,colo
r='magenta')
plt.show()
```



This plot explores the different possible number of channels for the convolutional layer. It is evident that having a higher number of channels results in better dev accuracy.

3.3 Best Model Sweep

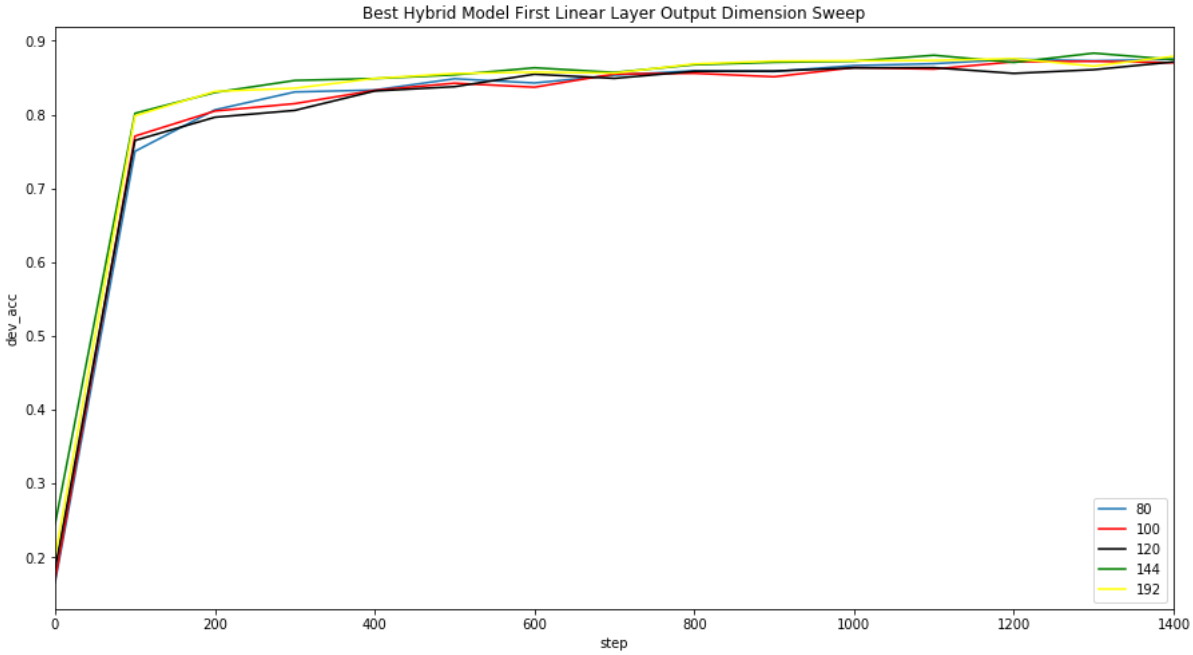
We pick two hyperparameters in our model and perform separate sweeps on them.

3.3.1 Hyperparameter 1

First Linear Layer Output Dimension

In [19]:
```
best_metrics_0 = pd.read_csv('best00-logs.csv')
best_metrics_1 = pd.read_csv('best01-logs.csv')
best_metrics_2 = pd.read_csv('best02-logs.csv')
best_metrics_3 = pd.read_csv('best03-logs.csv')
best_metrics_4 = pd.read_csv('best04-logs.csv')
```

In [20]:
```
plt.figure(figsize=(15, 8))
plt.title("Best Hybrid Model First Linear Layer Output Dimension Sweep")
ax = plt.gca()
plt.ylabel('dev_acc')
best_metrics_0.plot(kind='line',x='step',y='dev_acc',ax=ax,label=80)
best_metrics_1.plot(kind='line',x='step',y='dev_acc',ax=ax,label=100,col
or='red')
best_metrics_2.plot(kind='line',x='step',y='dev_acc',ax=ax,label=120,col
or='black')
best_metrics_3.plot(kind='line',x='step',y='dev_acc',ax=ax,label=144,col
or='green')
best_metrics_4.plot(kind='line',x='step',y='dev_acc',ax=ax,label=192,col
or='yellow')
plt.show()
```



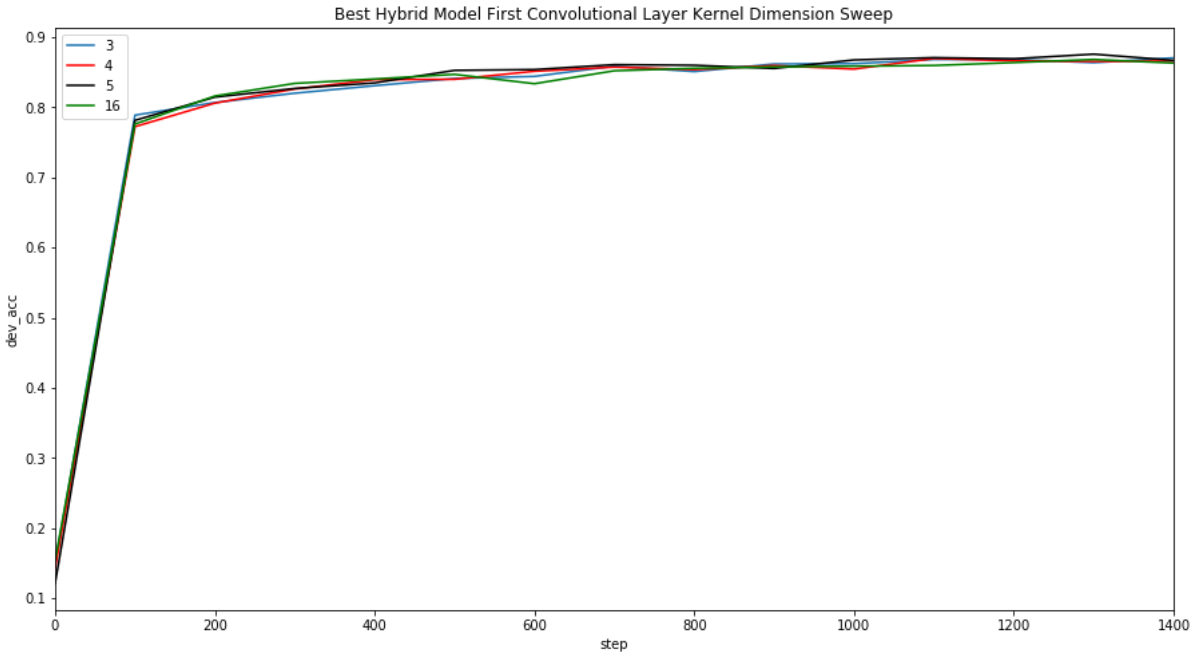
This hyperparameter dictates the size of the representation generated by the first linear layer (which has to be condensed by the second one). It looks like training is relatively robust to changes in this hyperparameter. However, we picked 120, the graph of which looks to be the highest and also trending upwards at step 1400.

3.3.2 Hyperparameter 2

First Convolutional Layer Kernel Dimension

In [21]:
```
best_metrics_0 = pd.read_csv('best10-logs.csv')
best_metrics_1 = pd.read_csv('best11-logs.csv')
best_metrics_2 = pd.read_csv('best12-logs.csv')
best_metrics_3 = pd.read_csv('best13-logs.csv')
```

In [22]:
```
plt.figure(figsize=(15, 8))
plt.title("Best Hybrid Model First Convolutional Layer Kernel Dimension
 Sweep")
ax = plt.gca()
plt.ylabel('dev_acc')
best_metrics_0.plot(kind='line',x='step',y='dev_acc',ax=ax,label=3)
best_metrics_1.plot(kind='line',x='step',y='dev_acc',ax=ax,label=4,color
='red')
best_metrics_2.plot(kind='line',x='step',y='dev_acc',ax=ax,label=5,color
='black')
best_metrics_3.plot(kind='line',x='step',y='dev_acc',ax=ax,label=16,colo
r='green')
plt.show()
```



Model performance seems to be robust to changes in kernel size of the first convolutional layer. Our choice of kernel size is 5 for the best model. During the sweep, we found that 1 and 6 turned out to be invalid values for this hyperparameter.

In [ ]: