The Payne

For this particular case, we generate ~160 training spectra and ~160 testing spectra using B. Kurucz codes and P. Cargile line list. The Teff and log g are drawn from the MIST isochrones. We consider stellar evolution from the main sequence stars up to the red clump stars with Teff from 3500 K to 7000 K. We also consider a wide range of metallicity with [Fe/H] = -2 to 0.5.

The following will show that using only 160 models, The Payne is an extremely precise generative model for this wide range of stellar parameters.

> Train neural network

This training step is quite slow. Training a single wavelength pixel typically takes 10 minutes to 1 hour. It requires running on a cluster. If you want to try this part on a personal laptop, make sure that you only train spectra with a few wavelength pixels. Note that although the training is excruciating slow, once the network is trained, fitting spectra (i.e. testing step) is very efficient.

In [ ]:
```
# import python packages
import numpy as np
from multiprocessing import Pool

import theano
import theano.tensor as T
from theano.tensor.nnet import sigmoid


#============================================================================================
# number of CPUs for parallel computing
num_CPU = 4

#--------------------------------------------------------------------------------------------
### some parameters of neural network ###
# how many portions will we split the training data to perform stochastic gradient descent
# smaller batch size (i.e. larger number) will take longer to converge
```

```
# smaller batch size (i.e. larger number) will take longer to converge
# but might converge with a smaller number of steps
mini_batch_size = 2

# initial step size in stochastic gradient descent
# smaller step size will be slower but will provide better convergence
eta_choice = 0.1

# the minimum step size beyond which we will truncate
min_eta = 0.001

# how many steps of gradient descent per loop are we going to perform
num_epochs_choice = 10000

# truncation criterion
trunc_diff = 0.003

# maximum number of loop (to avoid infinite loop)
# i.e. max_iter*num_epochs_choice is the maximum number of steps beyond which we will truncate
max_iter = 100

# how many neurons per layer
# here we always consider two fully connected layers
n_neurons = 10

#------------------------------------------------------------------------------------------
# define activation function that we will use in the validation step
# make sure this function is consistent with the training function
# here we choose a sigmoid function
def act_func(z):
    return 1.0/(1.0+np.exp(-z))


#==========================================================================================
# restore all spectra
# the variable "spectra" has a dimension of (# pixels, # spectra)
# the variable "labels" has a dimension of (# labels, # spectra)
# for this particular training set, the labels are (teff[K], logg, Fe/H)
```

```
# for this particular training set, the labels are (teff[k], logg, feh)
temp = np.load("kurucz_spectra.npz")
spectra = temp["Y_u_all"].T
labels = temp["labels_array"]

# only use half of them to train the neural network
# we will save the other half for testing
spectra = spectra[:,::2]
labels = labels[:,::2]

#-----------------------------------------------------------------------------------------
# neural networks typically train a function mapping from [0,1] -> [0,1]
# so here we scale both input (labels) and output (fluxes) to [0.1,0.9]

# record how we scale the labels
x_max = np.max(labels, axis=1)
x_min = np.min(labels, axis=1)

# scale the labels
labels = ((labels.T - x_min)*0.8/(x_max-x_min) + 0.1).T

# scale the fluxes
# here we assume normalized spectra
# for flux spectra this scaling has to be changed
spectra = spectra*0.8 + 0.1


#=========================================================================================
# theano is a very powerful package to train neural network
# it performs "auto diff", i.e., provides analytic differentiation of any cost function

# convert labels into a theano variable
training_x = theano.shared(np.asarray(labels.T, dtype='float64'))

#-----------------------------------------------------------------------------------------
# main network class
class Network(object):
    def __init__(self, layers, mini_batch_size):
```

```
        # initiate network properties
        self.layers = layers
        self.mini_batch_size = mini_batch_size
        self.params = [param for layer in self.layers for param in layer.params]

        self.x = T.dmatrix("x")
        self.y = T.dmatrix("y")

        init_layer = self.layers[0]
        init_layer.set_inpt(self.x, self)
        for j in xrange(1, len(self.layers)):
            prev_layer, layer = self.layers[j-1], self.layers[j]
            layer.set_inpt(prev_layer.output, self)
        self.output = self.layers[-1].output

        # create a property to record the cost function at each training step
        self.cost_train = []

#-----------------------------------------------------------------------------------------
    # stochastic gradient descent
    def SGD(self, training_x, training_y, epochs, eta):

        # reset the cost for each training loop
        self.cost_train = []

        # compute mini batches
        num_sample = training_x.get_value().shape[0]
        num_training_batches = num_sample/self.mini_batch_size

        # define cost function, symbolic gradients, and updates
        cost = self.layers[-1].cost(self)
        grads = T.grad(cost, self.params)
        updates = [(param, param-eta/self.mini_batch_size*grad)\
                       for param, grad in zip(self.params, grads)]

        i = T.lscalar()
```

```
        # randomize the training data for stochastic gradient descent
        ind = np.arange(num_sample)
        np.random.shuffle(ind)
        ind = theano.shared(ind)

        # define function to train a mini-batch
        train_mb = theano.function([i], cost, updates=updates,
            givens={self.x: training_x[ind[i*self.mini_batch_size:(i+1)*self.mini_batch_size]],
                    self.y: training_y[ind[i*self.mini_batch_size:(i+1)*self.mini_batch_size]]})

        # the actual training
        for epoch in xrange(epochs):
            cost_train_ij = 0.
            for minibatch_index in xrange(num_training_batches):
                # sum up all cost for each mini batch
                cost_train_ij += train_mb(minibatch_index)
            self.cost_train.append(cost_train_ij)

#-----------------------------------------------------------------------------------------
# define fully connected layers
### here we choose sigmoid function to be the activation function ###
class FullyConnectedLayer(object):
    def __init__(self, n_in, n_out, activation_fn=sigmoid):
        self.n_in = n_in
        self.n_out = n_out
        self.activation_fn = activation_fn

        # initialize weights and biases of the neural network
        self.w = theano.shared(np.asarray(np.random.normal(
                    loc=0.0, scale=np.sqrt(1.0/n_out), size=(n_in,n_out)),
                    dtype=theano.config.floatX))
        self.b = theano.shared(np.asarray(np.random.normal(\
                    loc=0.0, scale=1.0, size=(n_out,)),
                    dtype=theano.config.floatX))
        self.params = [self.w,self.b]
```

```
#-------------------------------------------------------------------------------------------
    # define input and output for each neural network layer
    def set_inpt(self, inpt, net):
        self.inpt = inpt.reshape((net.mini_batch_size, self.n_in))
        self.output = self.activation_fn(T.dot(self.inpt, self.w) + self.b)

    ### define a cost function ###
    def cost(self,net):
        return T.sum(T.abs_(net.y-self.output))


#=============================================================================================
# define training function for each wavelength pixel to run in parallel
# note we create individual neural network for each wavelength pixel
def train_pixel(pixel_no):

    # extract flux of a wavelength pixel
    training_y = theano.shared(np.asarray(np.array([spectra[pixel_no,:]]).T, dtype='float64'))

    # define the network
    net = Network([
        FullyConnectedLayer(n_in=training_x.get_value().shape[1], n_out=n_neurons),\
        FullyConnectedLayer(n_in=n_neurons, n_out=training_y.get_value().shape[1])],\
                  mini_batch_size)

#-------------------------------------------------------------------------------------------
    # initiate loop counter and step size
    loop_count = 0
    step_devide = 1.

    # sometimes the network can stuck at the initial point
    # so first we train for 1000 steps
    net.SGD(training_x, training_y, 1000, eta_choice)

    # we evaluate if the cost has improved
    while (np.abs(np.mean(net.cost_train[:100]) \
                    - np.mean(net.cost_train[-100:]))/(np.mean(net.cost_train[:100])) < 0.1)\
```

```
            and (loop_count < max_iter):

        # if not we reset the newtwork (and hence the initial point)
        # and loop until it finds a valid initial point
        net = Network([
                FullyConnectedLayer(n_in=training_x.get_value().shape[1], n_out=n_neurons),\
                FullyConnectedLayer(n_in=n_neurons, n_out=training_y.get_value().shape[1])],\
                          mini_batch_size)
        net.SGD(training_x, training_y, 1000, eta_choice)

        # increase counter
        loop_count += 1

#-----------------------------------------------------------------------------------------
    # after a good initial point is found, we proceed to the extensive training

    # initiate the deviation truncation criterion
    med_deviate = 1000.

    # loop until the deviation is smaller than the chosen truncation criterion
    # we also truncate if the step size has become too small
    while (med_deviate > trunc_diff) and (loop_count < max_iter) and (eta_choice/step_devide > min_e

        # continue to train the network if it has not converged yet
        net.SGD(training_x, training_y, num_epochs_choice, eta_choice/step_devide)

        # increase counter to avoid infinite loop
        loop_count += 1

        # check if the current step size is too large, i.e. cost does not change much
        if np.abs(np.mean(net.cost_train[:100])   \
                  - np.mean(net.cost_train[-100:]))/(np.mean(net.cost_train[:100])) < 0.01:

            # if so, we make the step size smaller
            step_devide = step_devide*2.

#-----------------------------------------------------------------------------------------
        # this is the validation step
```

```
        # this is the validation step
        # calcluate the deviation between the analytic approximation vs. the training models
        # in principle, we should consider validtion models here
        w_array_0 = net.layers[0].w.get_value().T
        b_array_0 = net.layers[0].b.get_value()
        w_array_1 = net.layers[1].w.get_value()[:,0]
        b_array_1 = net.layers[1].b.get_value()[0]

        predict_flux = act_func(np.sum(w_array_1*act_func(np.dot(w_array_0,labels).T\
                            + b_array_0), axis=1) + b_array_1)

        # remember to scale back the fluxes to the normal metric
        ### here we choose the maximum absolute deviation to be the truncation criterion ###
        med_deviate = np.max(np.abs((predict_flux - spectra[pixel_no,:])/0.8))

    # return the trained network for this pixel
    return net


#================================================================================================
# train in parallel for all wavelength pixels
import time
start_time = time.time()
pool = Pool(num_CPU)
net_array = pool.map(train_pixel,range(spectra.shape[0]))
print time.time() - start_time

#------------------------------------------------------------------------------------------------
# extract neural network parameters as numpy array
# the first layer
w_array_0 = np.array([net_array[i].layers[0].w.get_value().T\
                      for i in range(spectra.shape[0])])
b_array_0 = np.array([net_array[i].layers[0].b.get_value()\
                      for i in range(spectra.shape[0])])

# the second layer
w_array_1 = np.array([net_array[i].layers[1].w.get_value()[:,0]\
                      for i in range(spectra.shape[0])])
```

```
b_array_1 = np.array([net_array[i].layers[1].b.get_value()[0]\
                      for i in range(spectra.shape[0])])

#-----------------------------------------------------------------------------------------
# save the neural network parameters and record how we have scaled the labels
np.savez("neural_network.npz",\
         w_array_0=w_array_0, w_array_1=w_array_1,\
         b_array_0=b_array_0, b_array_1=b_array_1,\
         x_max=x_max, x_min=x_min)
```

Check how well we can predict the fluxes of spectra.

Ab-initio calculated spectrum vs. neural network reconstruction of a spectrum.

In [1]:
```
# import python packages
%matplotlib inline
import matplotlib.pyplot as plt
from matplotlib import rcParams
import numpy as np

#-----------------------------------------------------------------------------------------
# define plot properties
def rgb(r,g,b):
    return (float(r)/256.,float(g)/256.,float(b)/256.)

cb2 = [rgb(31,120,180), rgb(255,127,0), rgb(51,160,44), rgb(227,26,28), \
       rgb(166,206,227), rgb(253,191,111), rgb(178,223,138), rgb(251,154,153)]
rcParams['lines.linewidth'] = 1
rcParams['axes.color_cycle'] = cb2
rcParams['font.family'] = 'Bitstream Vera Sans'
rcParams['font.size'] = 35
```

```
#================================================================================================
# restore trained neural network
temp = np.load("neural_network.npz")

w_array_0 = temp["w_array_0"]
w_array_1 = temp["w_array_1"]
b_array_0 = temp["b_array_0"]
b_array_1 = temp["b_array_1"]
x_min = temp["x_min"]
x_max = temp["x_max"]

# activation function
# make sure this function is consistent with the trained network
def act_func(z):
    return 1.0/(1.0+np.exp(-z))


#================================================================================================
# restore all spectra
temp = np.load("kurucz_spectra.npz")
wavelength = temp["wavelength"]
spectra = temp["Y_u_all"].T
labels = temp["labels_array"]

#------------------------------------------------------------------------------------------------
### testing spectra (not used in the training steps) ###
spectra_testing = spectra[:,1::2]
labels_testing = labels[:,1::2]

# scale labels as before
labels_testing = ((labels_testing.T - x_min)*0.8/(x_max-x_min) + 0.1).T

# predict flux using the neural network
predict_flux_array = []
```

```
# for each of the spectra
for i in range(labels_testing.shape[1]):

    # we predict the flux ...
    predict_flux_array.append(act_func(np.sum(w_array_1*(act_func(np.dot(w_array_0,\
                    labels_testing[:,i]) + b_array_0)), axis=1) + b_array_1))

# convert into a numpy array
predict_testing = np.array(predict_flux_array).T

# recall that the predicted fluxes are scaled to [0.1,0.9]
# so here we have to rescale it back to the normal metric
predict_testing = (predict_testing - 0.1)/0.8

#-------------------------------------------------------------------------------------------
### we perform the same for training spectra ###
spectra_training = spectra[:,0::2]
labels_training = labels[:,0::2]
labels_training = ((labels_training.T - x_min)*0.8/(x_max-x_min) + 0.1).T
predict_flux_array = []
for i in range(labels_training.shape[1]):
    predict_flux_array.append(act_func(np.sum(w_array_1*(act_func(np.dot(w_array_0,\
                    labels_training[:,i]) + b_array_0)), axis=1) + b_array_1))
predict_training = np.array(predict_flux_array).T
predict_training = (predict_training - 0.1)/0.8


#===========================================================================================
# initiate the plot
fig = plt.figure(figsize=[22,50]);
ax = fig.add_subplot(111)
ax.spines['top'].set_color('none')
ax.spines['bottom'].set_color('none')
ax.spines['left'].set_color('none')
ax.spines['right'].set_color('none')
ax.tick_params(labelcolor='w', top='off', bottom='off', left='off', right='off')
```

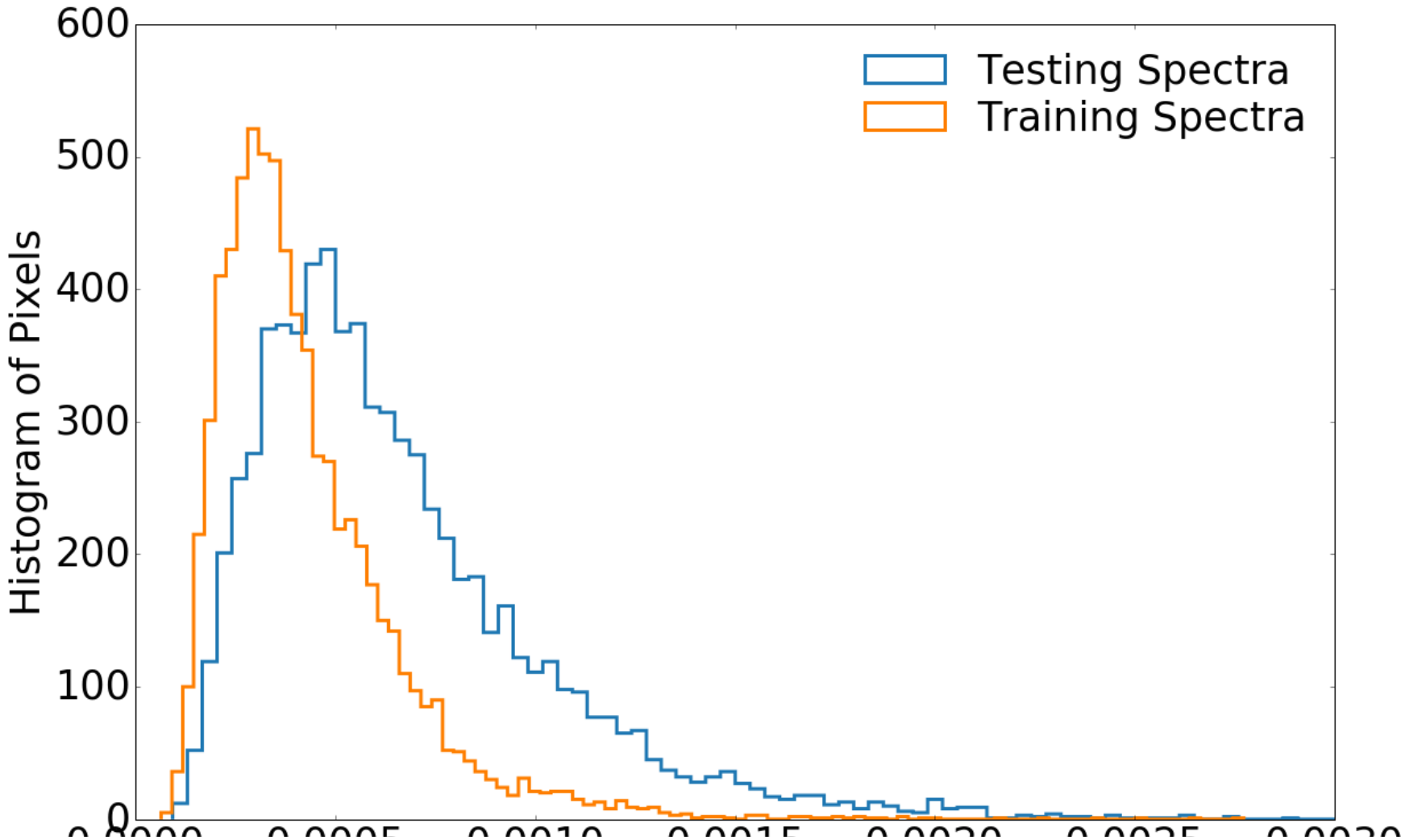
```
# axis labels
ax.set_xlabel("Wavelength [nm]", labelpad=30, fontsize=50);
ax.set_ylabel("Normalized Flux", labelpad=30, fontsize=50);

#-----------------------------------------------------------------------------------------
# break wavelength into different subplots
for i in range(8):
    ax = fig.add_subplot(8,1,i+1)
    plt.xlim([1500+i*25.,1500+(i+1)*25])
    plt.ylim([0.5,1.1])
    ax.tick_params(axis='x', pad=20);
    plt.plot(wavelength/10, spectra_testing[:,0],\
             color=cb2[0], lw=2, label="Ab-Initio Calculation")
    plt.plot(wavelength/10, predict_testing[:,0],\
             color=cb2[3], lw=2, label="The Payne Reconstruction", alpha=0.8)

#-----------------------------------------------------------------------------------------
# plot legend
plt.legend(loc="lower right", fontsize=35, frameon=False,\
               borderpad=0.2, labelspacing=0.2, scatterpoints=1)

# save figure
plt.tight_layout()
plt.savefig("flux_prediction.png")
```

Evaluate the median absolute deviation of all wavelength pixels.

In [2]:
```
# initiate plot
fig = plt.figure(figsize=[18,12]);
ax = fig.gca();

# axis labels
ax.set_xlabel(r"Median Absolute Deviation");
ax.set_ylabel(r"Histogram of Pixels");

# plotting range
plt.xlim([0,0.003])

#-------------------------------------------------------------------------------------------
# ploting histogram
plt.hist(np.median(np.abs(spectra_testing-predict_testing), axis=1), bins=100,\
         histtype="step", lw=3, label="Testing Spectra", )
plt.hist(np.median(np.abs(spectra_training-predict_training), axis=1), bins=100,\
         histtype="step", lw=3, label="Training Spectra")

print np.median(np.abs(spectra_testing-predict_testing), axis=1).shape

#-------------------------------------------------------------------------------------------
# plot legend
plt.legend(loc="upper right", fontsize=35, frameon=False,\
```

```
                borderpad=0.2, labelspacing=0.2, scatterpoints=1)

# save figure
plt.tight_layout()
plt.savefig("flux_deviation.png")
```

```
(7214,)
```



Fitting spectra with the trained network.

In [5]:
```
# import python packages
import numpy as np
from multiprocessing import Pool
from scipy.optimize import curve_fit


#========================================================================================
# number of processors for parallel computing
num_CPU = 4

#----------------------------------------------------------------------------------------
# restore testing spectra
temp = np.load("kurucz_spectra.npz")
spectra = temp["Y_u_all"].T
labels = temp["labels_array"]
spectra_testing = spectra[:,1::2]
labels_testing = labels[:,1::2]

# scale the flux to the neural network scale
# if the fluxes have uncertainties, also remember to scale the uncertainties
spectra_testing = spectra_testing*0.8 + 0.1

#----------------------------------------------------------------------------------------
# restore trained neural network
temp = np.load("neural_network.npz")
w_array_0 = temp["w_array_0"]
w_array_1 = temp["w_array_1"]
b_array_0 = temp["b_array_0"]
```

```
b_array_0 = temp["b_array_0"]
b_array_1 = temp["b_array_1"]
x_min = temp["x_min"]
x_max = temp["x_max"]


#================================================================================
# activation function
# make sure this function is consistent with the trained network
def act_func(z):
    return 1.0/(1.0+np.exp(-z))

# predict spectrum
def predict_spec(input_param, *labels):
    predict_flux = act_func(np.sum(w_array_1*(act_func(np.dot(w_array_0,labels)\
                                            + b_array_0)), axis=1) + b_array_1)
    return predict_flux

# define function to perform testing steps in batch (over many spectra)
def fit_spectrum(spec_no):
    try:
        # here we set the noise (sigma) to be 0.01 dex
        # but note that we assume noiseless spectra in our testing case
        # here we also start with the right initial point
        # in practice multiple resets might be needed
        popt, pcov = curve_fit(predict_spec, [spec_no], spectra_testing[:,spec_no],\
                               p0 = (labels_testing[:,spec_no]-x_min)*0.8/(x_max-x_min) + 0.1,\
                               sigma=0.01,\
                               absolute_sigma=True, bounds=(0,1))
    except:
        # if the minimization does not converge, return some nul values
        popt = np.zeros(labels_testing.shape[0]) - 9999.
    return popt

#--------------------------------------------------------------------------------
# fit spectra in parallel
pool = Pool(num_CPU)
best_fit_results = np.array(pool.map(fit_spectrum,range(spectra_testing.shape[1]))).T
```

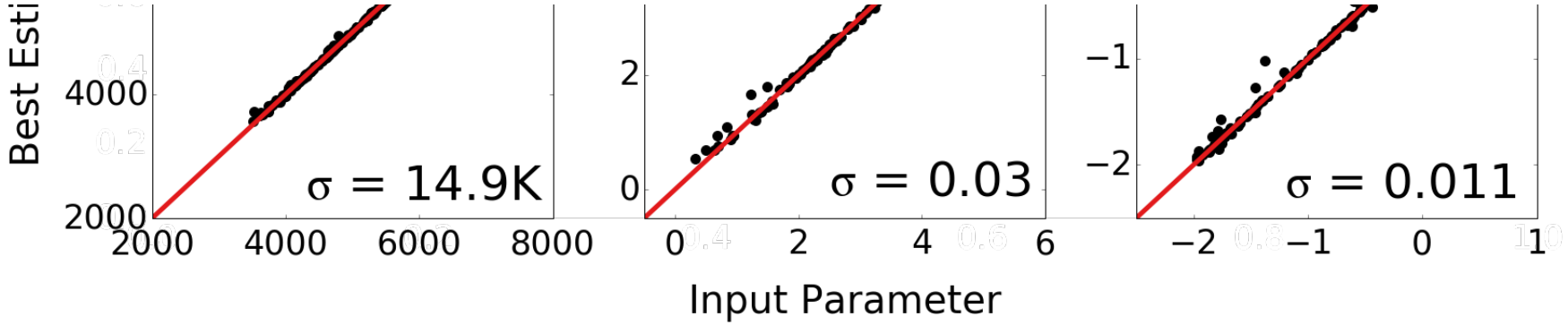
```
# recall that we have scaled the labels to train the network
# here we rescale the best-fitting results back to the original scale
best_fit_results = ((best_fit_results.T-0.1)*(x_max-x_min)/0.8+x_min).T
```

Plot the best fitting results vs. input parameters

In [4]:
```
# import python packages
%matplotlib inline
import matplotlib.pyplot as plt
from matplotlib import rcParams
import numpy as np

#-------------------------------------------------------------------------------------------
# define plot properties
def rgb(r,g,b):
    return (float(r)/256.,float(g)/256.,float(b)/256.)

cb2 = [rgb(31,120,180), rgb(255,127,0), rgb(51,160,44), rgb(227,26,28), \
       rgb(166,206,227), rgb(253,191,111), rgb(178,223,138), rgb(251,154,153)]
rcParams['lines.linewidth'] = 1
rcParams['axes.color_cycle'] = cb2
rcParams['font.family'] = 'Bitstream Vera Sans'
rcParams['font.size'] = 35


#===========================================================================================
# initiate the plot
fig = plt.figure(figsize=[25,9]);
ax = fig.add_subplot(111)
ax.spines['top'].set_color('none')
ax.spines['bottom'].set color('none')
```

```
ax.spines['left'].set_color('none')
ax.spines['right'].set_color('none')
ax.tick_params(labelcolor='w', top='off', bottom='off', left='off', right='off')

# axis labels
ax.set_xlabel("Input Parameter", fontsize=40, labelpad=30);
ax.set_ylabel("Best Estimate", fontsize=40, labelpad=50);

#-----------------------------------------------------------------------------------------
# label name
label_name = ["$\mathregular{T_{eff}\;[K]}$",\
              "$\mathregular{log\,g}$",\
              "[Fe/H]"]

#-----------------------------------------------------------------------------------------
# loop over all labels
for u1 in range(len(label_name)):
    ax = fig.add_subplot(1, 3, u1+1)
    ax.tick_params(axis='x', pad=10);
    plt.locator_params(nbins=5)

    # plot results
    plt.scatter(labels_testing[u1,:], best_fit_results[u1,:], color="black", s=100)

#-----------------------------------------------------------------------------------------
    # plotting parameters
    # for Teff
    if u1 == 0:
        # plotting range
        plt.xlim([2000,8000])
        plt.ylim([2000,8000])

        # plot 1-to-1 line guide line
        plt.plot([2000,8000],[2000,8000], lw=5, color=cb2[3])

        # plot the name of the label
        plt.text(2200, 7000, label_name[u1], fontsize=50)
```

```
        # calculate and write the 1 sigma error
        plt.text(4300, 2300, r"$\mathregular{\sigma}$ = "\
            + "%.1f" % (np.percentile(labels_testing[u1,:]-best_fit_results[u1,:], 68.3)\
                - np.percentile(labels_testing[u1,:]-best_fit_results[u1,:], 31.7)) + "K", fontsize=

    # for logg
    if u1 == 1:
        plt.xlim([-0.5,6.0])
        plt.ylim([-0.5,6.0])
        plt.plot([-0.5,6.0],[-0.5,6.0], lw=5, color=cb2[3])
        plt.text(-0.1, 4.9, label_name[u1], fontsize=50)
        plt.text(2.5, -0.1, r"$\mathregular{\sigma}$ = "\
            + "%.2f" % (np.percentile(labels_testing[u1,:]-best_fit_results[u1,:], 68.3)\
                - np.percentile(labels_testing[u1,:]-best_fit_results[u1,:], 31.7)), fontsize=50)

    # for [Fe/H]
    if u1 == 2:
        plt.xlim([-2.5,1.0])
        plt.ylim([-2.5,1.0])
        plt.plot([-2.5,1.0],[-2.5,1.0], lw=5, color=cb2[3])
        plt.text(-2.3, 0.4, label_name[u1], fontsize=50)
        plt.text(-1.2, -2.3, r"$\mathregular{\sigma}$ = "\
            + "%.3f" % (np.percentile(labels_testing[u1,:]-best_fit_results[u1,:], 68.3)\
                - np.percentile(labels_testing[u1,:]-best_fit_results[u1,:], 31.7)), fontsize=50)

#------------------------------------------------------------------------------
# save figure
plt.tight_layout(w_pad=0.7,h_pad=0.7)
plt.savefig("fitting_results.png")
```


Best Esti
4000
2000
σ = 14.9K
2000
4000
6000
8000
2
0
σ = 0.03
0
2
4
6
−1
−2
σ = 0.011
−2
−1
0
1
Input Parameter

In [ ]: