```
%matplotlib inline
```

Multi layer perceptron from scratch - CIFAR10 dataset

Reference: https://towardsdatascience.com/building-neural-network-from-scratch-9c88535bf8e9

A neural network needs a few building blocks

- Dense layer - a fully-connected layer, $f(X) = W \cdot X + \vec{b}$
- ReLU layer (activation function to introduce non-linearity)
- Loss function (crossentropy in case of multi-class classification problem)
- Backprop algorithm - a stochastic gradient descent with backpropageted gradients

Let's approach them one at a time.

Let's start by importing some libraires required for creating our neural network.

```
from __future__ import print_function
import numpy as np ## For numerical python
np.random.seed(42)
```

Every layer will have a forward pass and backpass implementation. Let's create a main class layer wh Backward pass .backward().

```
class Layer:
    """
    A building block. Each layer is capable of performing two things:

    - Process input to get output:           output = layer.forward(input)

    - Propagate gradients through itself:    grad_input = layer.backward(input, grad_output)

    Some layers also have learnable parameters which they update during layer.backward.
    """
    def __init__(self):
        """Here we can initialize layer parameters (if any) and auxiliary stuff."""
        # A dummy layer does nothing
        pass

    def forward(self, input):
        """
        Takes input data of shape [batch, input_units], returns output data [batch, output_un
        """
        # A dummy layer just returns whatever it gets as input.
        return input
```

```
    def backward(self, input, grad_output):
        """
        Performs a backpropagation step through the layer, with respect to the given input.

        To compute loss gradients w.r.t input, we need to apply chain rule (backprop):

        d loss / d x  = (d loss / d layer) * (d layer / d x)

        Luckily, we already receive d loss / d layer as input, so you only need to multiply i

        If our layer has parameters (e.g. dense layer), we also need to update them here usin
        """
        # The gradient of a dummy layer is precisely grad_output, but we'll write it more exp
        num_units = input.shape[1]

        d_layer_d_input = np.eye(num_units)

        return np.dot(grad_output, d_layer_d_input) # chain rule
```

Nonlinearity ReLU layer

This is the simplest layer you can get: it simply applies a nonlinearity to each element of your network

```
class ReLU(Layer):
    def __init__(self):
        """ReLU layer simply applies elementwise rectified linear unit to all inputs"""
        pass

    def forward(self, input):
        """Apply elementwise ReLU to [batch, input_units] matrix"""
        relu_forward = np.maximum(0,input)
        return relu_forward

    def backward(self, input, grad_output):
        """Compute gradient of loss w.r.t. ReLU input"""
        relu_grad = input > 0
        return grad_output*relu_grad
```

Dense layer

Now let's build something more complicated. Unlike nonlinearity, a dense layer actually has somethin

A dense layer applies affine transformation. In a vectorized form, it can be described as:

$$f(X) = W \cdot X + \vec{b}$$

Where

- X is an object-feature matrix of shape [batch_size, num_features],

- W is a weight matrix [num_features, num_outputs]
- and b is a vector of num_outputs biases.

Both W and b are initialized during layer creation and updated each time backward is called. Note that a trick to train our model to converge faster read more. Instead of initializing our weights with small n initialize our weights with mean zero and variance of 2/(number of inputs + number of outputs)

```
class Dense(Layer):
    def __init__(self, input_units, output_units, learning_rate=0.1):
        """
        A dense layer is a layer which performs a learned affine transformation:
        f(x) = <W*x> + b
        """
        self.learning_rate = learning_rate
        self.weights = np.random.normal(loc=0.0,
                                        scale = np.sqrt(2/(input_units+output_units)),
                                        size = (input_units,output_units))
        self.biases = np.zeros(output_units)

    def forward(self,input):
        """
        Perform an affine transformation:
        f(x) = <W*x> + b

        input shape: [batch, input_units]
        output shape: [batch, output units]
        """
        return np.dot(input,self.weights) + self.biases

    def backward(self,input,grad_output):
        # compute d f / d x = d f / d dense * d dense / d x
        # where d dense/ d x = weights transposed
        grad_input = np.dot(grad_output, self.weights.T)

        # compute gradient w.r.t. weights and biases
        grad_weights = np.dot(input.T, grad_output)
        grad_biases = grad_output.mean(axis=0)*input.shape[0]

        assert grad_weights.shape == self.weights.shape and grad_biases.shape == self.biases.

        # Here we perform a stochastic gradient descent step.
        self.weights = self.weights - self.learning_rate * grad_weights
        self.biases = self.biases - self.learning_rate * grad_biases

        return grad_input
```

The loss function

Since we want to predict probabilities, it would be logical for us to define softmax nonlinearity on top predicted probabilities. However, there is a better way to do so.

If we write down the expression for crossentropy as a function of softmax logits (a), you'll see:

$$loss = -log \frac{e^{a_{correct}}}{\sum_i e^{a_i}}$$

If we take a closer look, we'll see that it can be rewritten as:

$$loss = -a_{correct} + log \sum_i e^{a_i}$$

It's called Log-softmax and it's better than naive log(softmax(a)) in all aspects:

- Better numerical stability
- Easier to get derivative right
- Marginally faster to compute

```
def softmax_crossentropy_with_logits(logits,reference_answers):
    """Compute crossentropy from logits[batch,n_classes] and ids of correct answers"""
    logits_for_answers = logits[np.arange(len(logits)),reference_answers]

    xentropy = - logits_for_answers + np.log(np.sum(np.exp(logits),axis=-1))

    return xentropy

def grad_softmax_crossentropy_with_logits(logits,reference_answers):
    """Compute crossentropy gradient from logits[batch,n_classes] and ids of correct answers"
    ones_for_answers = np.zeros_like(logits)
    ones_for_answers[np.arange(len(logits)),reference_answers] = 1

    softmax = np.exp(logits) / np.exp(logits).sum(axis=-1,keepdims=True)

    return (- ones_for_answers + softmax) / logits.shape[0]
```

Full network

Now let's combine what we've just built into a working neural network. We are going to use CIFAR10 d already have it in the numpy array format, so let's import it!.

```
import torch
import torchvision
import torchvision.transforms as transforms

def load_dataset(flatten=False):
    transform = transforms.Compose(
    [transforms.ToTensor(),
     transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))])
```

```
    trainset = torchvision.datasets.CIFAR10(root='./data', train=True,
                                            download=True, transform=None)
    trainloader = torch.utils.data.DataLoader(trainset, batch_size=4,
                                              shuffle=True, num_workers=2)

    testset = torchvision.datasets.CIFAR10(root='./data', train=False,
                                           download=True, transform=transform)
    testloader = torch.utils.data.DataLoader(testset, batch_size=4,
                                             shuffle=False, num_workers=2)

    classes = ('plane', 'car', 'bird', 'cat',
               'deer', 'dog', 'frog', 'horse', 'ship', 'truck')

    X_train = trainset.data
    y_train = trainset.targets
    X_test = testset.data
    y_test = testset.targets

    # normalize x
    X_train = X_train.astype(float) / 255
    X_test = X_test.astype(float) / 255

    # we reserve the last 10000 training examples for validation
    X_train, X_val = X_train[:-10000], X_train[-10000:]
    y_train, y_val = y_train[:-10000], y_train[-10000:]

    if flatten:
        X_train = X_train.reshape([X_train.shape[0], -1])
        X_val = X_val.reshape([X_val.shape[0], -1])
        X_test = X_test.reshape([X_test.shape[0], -1])

    return X_train, y_train, X_val, y_val, X_test, y_test, classes, trainloader, testloader

X_train, y_train, X_val, y_val, X_test, y_test, classes, trainloader, testloader = load_datas
```

```
Files already downloaded and verified
Files already downloaded and verified
[0.23137255 0.24313725 0.24705882 ... 0.48235294 0.36078431 0.28235294]
```

We'll define network as a list of layers, each applied on top of previous one. In this setting, computing

```
network = []
network.append(Dense(X_train.shape[1],100))
network.append(ReLU())
network.append(Dense(100,200))
network.append(ReLU())
network.append(Dense(200,10))


def forward(network, X):
    """
```

```
    Compute activations of all network layers by applying them sequentially.
    Return a list of activations for each layer.
    """
    activations = []
    input = X

    # Looping through each layer
    for l in network:
        activations.append(l.forward(input))
        # Updating input to last layer output
        input = activations[-1]

    assert len(activations) == len(network)
    return activations

def predict(network,X):
    """
    Compute network predictions. Returning indices of largest Logit probability
    """
    logits = forward(network,X)[-1]
    return logits.argmax(axis=-1)

def train(network,X,y):
    """
    Train our network on a given batch of X and y.
    We first need to run forward to get all layer activations.
    Then we can run layer.backward going from last to first layer.
    After we have called backward for all layers, all Dense layers have already made one grad
    """

    # Get the layer activations
    layer_activations = forward(network,X)
    layer_inputs = [X]+layer_activations  #layer_input[i] is an input for network[i]
    logits = layer_activations[-1]

    # Compute the loss and the initial gradient
    loss = softmax_crossentropy_with_logits(logits,y)
    loss_grad = grad_softmax_crossentropy_with_logits(logits,y)

    # Propagate gradients through the network
    # Reverse propogation as this is backprop
    for layer_index in range(len(network))[::-1]:
        layer = network[layer_index]

        loss_grad = layer.backward(layer_inputs[layer_index],loss_grad) #grad w.r.t. input, a

    return np.mean(loss)
```

Training loop

We split data into minibatches, feed each such minibatch into the network and update weights. This t
stochastic gradient descent.

```
from tqdm import trange
def iterate_minibatches(inputs, targets, batchsize, shuffle=False):
    assert len(inputs) == len(targets)
    if shuffle:
        indices = np.random.permutation(len(inputs))
    for start_idx in trange(0, len(inputs) - batchsize + 1, batchsize):
        if shuffle:
            excerpt = indices[start_idx:start_idx + batchsize]
        else:
            excerpt = slice(start_idx, start_idx + batchsize)
        yield inputs[excerpt], targets[excerpt]


def get_network(input_units, output_units, learning_rate=0.1, epochs=25, dense_output_units=[
    if print_network:
        print('\tNETWORK: Multi layer perceptron')
    network = []
    network.append(Dense(input_units, dense_output_units[0], learning_rate))
    if print_network:
        print('\t\tDense(input_units={}, output_units={}, learning_rate={})'.format(input_uni
    network.append(ReLU())
    if print_network:
        print('\t\tReLU()')

    for i, _ in enumerate(dense_output_units):
      if i == len(dense_output_units) - 1:
        break
      network.append(Dense(dense_output_units[i], dense_output_units[i+1], learning_rate))
      if print_network:
          print('\t\tDense(input_units={}, output_units={}, learning_rate={})'.format(dense_o
      network.append(ReLU())
      if print_network:
          print('\t\tReLU()')

    network.append(Dense(dense_output_units[-1], output_units))
    if print_network:
        print('\t\tDense(input_units={}, output_units={}, learning_rate={})'.format(dense_out

    return network


from IPython.display import clear_output
from time import time

def training_loop(network, input_units, output_units, learning_rate=0.1, epochs=25, dense_out
    if len(dense_output_units) < 2:
      return
```

```
    epoch_start = time()
    train_acc_list = []
    val_acc_list = []
    epoch_time_list = []

    last_train_accuracy = 0.0
    last_validation_accuracy = 0.0

    for epoch in range(epochs):

        for x_batch,y_batch in iterate_minibatches(X_train,y_train,batchsize=batchsize,shuffl
            train(network,x_batch,y_batch)

        train_acc_list.append(np.mean(predict(network,X_train)==y_train))
        val_acc_list.append(np.mean(predict(network,X_val)==y_val))

        clear_output()
        print("Epoch", epoch)
        print("Training accuracy: {:.2f}%".format(train_acc_list[-1]*100))
        print("Validation accuracy: {:.2f}%".format(val_acc_list[-1]*100))
        epoch_time = time() - epoch_start
        epoch_time_list.append(epoch_time)
        print("Epoch's processing time: {:.2f} seconds".format(epoch_time))
        plt.plot(train_acc_list, label='train accuracy')
        plt.plot(val_acc_list, label='val accuracy')
        plt.legend(loc='best')
        plt.grid()
        plt.show()

    return train_acc_list, val_acc_list, epoch_time_list
```

Testing different dense layers

- Using 2 dense layers with learning_rate=0.1 (100 and 200 output units). Minibatches: batchsize=
- Using 3 dense layers with learning_rate=0.1 (100, 200, and 300 output units). Minibatches: batch
- Using 4 dense layers with learning_rate=0.1 (100, 200, 300 and 400 output units). Minibatches: b
- Using 5 dense layers with learning_rate=0.1 (100, 200, 300, 400, and 500 output units). Minibatc

```
validation_accuracy_list = []


from time import time
import numpy as np

def mlp_training(network, input_units, output_units, learning_rate, epochs, dense_output_unit
    training_start = time()
    train_acc_list, val_acc_list, epoch_time_list = training_loop(network, input_units=input_
    print("Total time: {} seconds".format(time() - training_start))
```

```
        print("Total time: {} seconds".format(time() - training_start))

        labels = ['epoch {}'.format(str(i).zfill(3)) for i in range(epochs)]

        train_acc_np = np.asarray(train_acc_list)
        print("\nTraining accuracy list: {}".format(train_acc_np))
        print("Training accuracy (Mean +/- Std): %0.2f (+/- %0.2f)" % (train_acc_np.mean()*100, t
        # Plot horizontal bar
        values = [v * 100 for v in train_acc_list]
        plot_horizontal_bar(labels, values, xlabel='Accuracy', ylabel='', title='Training accurac

        val_acc_np = np.asarray(val_acc_list)
        print("\nValidation accuracy list: {}".format(val_acc_np))
        print("Validation accuracy (Mean +/- Std): %0.2f (+/- %0.2f)" % (val_acc_np.mean()*100, v

        # Save validation accuracy to plotting: Validation accuracy Vs Number of dense layers
        validation_accuracy_list.append(val_acc_np.mean()*100)

        # Plot horizontal bar
        values = [v * 100 for v in val_acc_list]
        plot_horizontal_bar(labels, values, xlabel='Accuracy', ylabel='', title='Validation accur

        epoch_time_np = np.asarray(epoch_time_list)
        print("\nEpoch time list: {}".format(epoch_time_np))
        print("Epoch time (Mean +/- Std): %0.2f (+/- %0.2f)" % (epoch_time_np.mean(), epoch_time_
        # Plot horizontal bar
        values = [v * 100 for v in epoch_time_list]
        plot_horizontal_bar(labels, values, xlabel='Time', ylabel='', title='Epoch time in second


    def plot_horizontal_bar(x, y, xlabel, ylabel, title, use_xlim=False):
        fig, ax = plt.subplots()
        width = 0.75 # the width of the bars
        ind = np.arange(len(y))  # the x locations for the groups
        ax.barh(ind, y, width, color="blue")
        ax.set_yticks(ind+width/2)
        ax.set_yticklabels(x, minor=False)
        plt.title(title)
        plt.xlabel(xlabel)
        plt.ylabel(ylabel)

        for i, v in enumerate(y):
            ax.text(v + 3, i + .25, '%0.2f'%(v), color='blue', fontweight='bold')

        if use_xlim:
            plt.xlim(0, 120)
            plt.tight_layout()

        plt.show()
```

Using **2 dense layers** (100 and 200 output units) with learning_rate=0.1 . Minibatc

```
dense_output_units=[100, 200]

network = get_network(input_units=X_train.shape[1], output_units=10, epochs=25, dense_output_
```

```
NETWORK: Multi layer perceptron
        Dense(input_units=3072, output_units=100, learning_rate=0.1)
        ReLU()
        Dense(input_units=100, output_units=200, learning_rate=0.1)
        ReLU()
        Dense(input_units=200, output_units=10, learning_rate=0.1)
```

```
# TODO FIX BUG
mlp_training(network, input_units=X_train.shape[1], output_units=10, learning_rate=0.1, epoch
```

```
  0%|          | 0/1250 [00:00<?, ?it/s]
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-37-6112eb8b39fa> in <module>()
----> 1 mlp_training(network, input_units=X_train.shape[1], output_units=10, learning_ra

                                 2 frames
<ipython-input-30-4e87c32f38ed> in iterate_minibatches(inputs, targets, batchsize, shuff
      9         else:
     10             excerpt = slice(start_idx, start_idx + batchsize)
---> 11         yield inputs[excerpt], targets[excerpt]

TypeError: only integer scalar arrays can be converted to a scalar index

SEARCH STACK OVERFLOW
```

Using **3 dense layers** (100, 200 and 300 output units) with learning_rate=0.1 . Mir shuffle=True

```
dense_output_units=[100, 200, 300]

network = get_network(input_units=X_train.shape[1], output_units=10, epochs=25, dense_output_


mlp_training(network, input_units=X_train.shape[1], output_units=10, learning_rate=0.1, epoch
```

```
  0%|          | 0/1250 [00:00<?, ?it/s]
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-18-6112eb8b39fa> in <module>()
----> 1 mlp_training(network, input_units=X_train.shape[1], output_units=10, learning_ra

 2 frames
<ipython-input-10-4e87c32f38ed> in iterate_minibatches(inputs, targets, batchsize, shuff
      9         else:
     10             excerpt = slice(start_idx, start_idx + batchsize)
---> 11         yield inputs[excerpt], targets[excerpt]

TypeError: only integer scalar arrays can be converted to a scalar index

SEARCH STACK OVERFLOW
```

Using **4 dense layers** (100, 200, 300 and 400 output units) with learning_rate=0.1 shuffle=True

```
dense_output_units=[100, 200, 300, 400]

network = get_network(input_units=X_train.shape[1], output_units=10, epochs=25, dense_output_

mlp_training(network, input_units=X_train.shape[1], output_units=10, learning_rate=0.1, epoch
```

```
  0%|          | 0/1250 [00:00<?, ?it/s]
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-19-6112eb8b39fa> in <module>()
----> 1 mlp_training(network, input_units=X_train.shape[1], output_units=10, learning_ra

 2 frames
<ipython-input-10-4e87c32f38ed> in iterate_minibatches(inputs, targets, batchsize, shuff
      9         else:
     10             excerpt = slice(start_idx, start_idx + batchsize)
---> 11         yield inputs[excerpt], targets[excerpt]

TypeError: only integer scalar arrays can be converted to a scalar index

SEARCH STACK OVERFLOW
```

Using **5 dense layers** (100, 200, 300, 400 and 500 output units) with learning_rate shuffle=True

```
dense_output_units=[100, 200, 300, 400, 500]
```

```
network = get_network(input_units=X_train.shape[1], output_units=10, epochs=25, dense_output_
```

```
mlp_training(network, input_units=X_train.shape[1], output_units=10, learning_rate=0.1, epoch
```

Best number of dense layers

- Highest validation accuracy (mean)

```
labels = ['2 dense layers', '3 dense layers', '4 dense layers', '5 dense layers']
plot_horizontal_bar(labels, validation_accuracy_list, xlabel='Validation accuracy', ylabel=''
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-20-f9455c56b59c> in <module>()
      1 labels = ['2 dense layers', '3 dense layers', '4 dense layers', '5 dense layers'
----> 2 plot_horizontal_bar(labels, validation_accuracy_list, xlabel='Validation accurac

<ipython-input-15-d205da539b63> in plot_horizontal_bar(x, y, xlabel, ylabel, title, use_
      1 def plot_horizontal_bar(x, y, xlabel, ylabel, title, use_xlim=False):
----> 2     fig, ax = plt.subplots()
      3     width = 0.75 # the width of the bars
      4     ind = np.arange(len(y))  # the x locations for the groups
      5     ax.barh(ind, y, width, color="blue")

NameError: name 'plt' is not defined

SEARCH STACK OVERFLOW
```

```
validation_accuracy_list = []
```

Testing different learning rates

- Using 3 dense layers with learning_rate = 0.1. Minibatches: batchsize=32, shuffle=True
- Using 3 dense layers with learning_rate = 0.01. Minibatches: batchsize=32, shuffle=True
- Using 3 dense layers with learning_rate = 0.001. Minibatches: batchsize=32, shuffle=True
- Using 3 dense layers with learning_rate = 0.0001. Minibatches: batchsize=32, shuffle=True

```
dense_output_units=[100, 200, 300]
```

```
network = get_network(input_units=X_train.shape[1], output_units=10, epochs=25, dense_output_
```

```
NETWORK: Multi layer perceptron
        Dense(input_units=3072, output_units=100, learning_rate=0.1)
        ReLU()
        Dense(input_units=100, output_units=200, learning_rate=0.1)
        ReLU()
        Dense(input_units=200, output_units=300, learning_rate=0.1)
        ReLU()
        Dense(input_units=300, output_units=10, learning_rate=0.1)
```

Using 3 dense layers with **learning_rate = 0.1**. Minibatches: batchsize=32, shuffle

```
mlp_training(network, input_units=X_train.shape[1], output_units=10, learning_rate=0.1, epoch
```

```
  0%|          | 0/1250 [00:00<?, ?it/s]
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-23-6112eb8b39fa> in <module>()
----> 1 mlp_training(network, input_units=X_train.shape[1], output_units=10, learning_ra

                                  2 frames
<ipython-input-10-4e87c32f38ed> in iterate_minibatches(inputs, targets, batchsize, shuff
      9         else:
     10             excerpt = slice(start_idx, start_idx + batchsize)
---> 11         yield inputs[excerpt], targets[excerpt]

TypeError: only integer scalar arrays can be converted to a scalar index
```

SEARCH STACK OVERFLOW

Using 3 dense layers with **learning_rate = 0.01**. Minibatches: batchsize=32, shuffl

```
mlp_training(network, input_units=X_train.shape[1], output_units=10, learning_rate=0.01, epoc
```

```
  0%|          | 0/1250 [00:00<?, ?it/s]
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-24-464fe393e384> in <module>()
----> 1 mlp_training(network, input_units=X_train.shape[1], output_units=10, learning_ra

                                  2 frames
<ipython-input-10-4e87c32f38ed> in iterate_minibatches(inputs, targets, batchsize, shuff
      9         else:
     10             excerpt = slice(start_idx, start_idx + batchsize)
---> 11         yield inputs[excerpt], targets[excerpt]

TypeError: only integer scalar arrays can be converted to a scalar index
```

SEARCH STACK OVERFLOW

Using 3 dense layers with **learning_rate = 0.001**. Minibatches: batchsize=32, shuf

```
mlp_training(network, input_units=X_train.shape[1], output_units=10, learning_rate=0.001, epo
```

Using 3 dense layers with **learning_rate = 0.0001**. Minibatches: batchsize=32, shu

```
mlp_training(network, input_units=X_train.shape[1], output_units=10, learning_rate=0.0001, ep
```

Best learning rate

- Highest validation accuracy (mean)

```
labels = ['lr=0.1', 'lr=0.01', 'lr=0.001', 'lr=0.0001']
plot_horizontal_bar(labels, validation_accuracy_list, xlabel='Validation accuracy', ylabel=''
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-25-db4ebf6ae3a3> in <module>()
      1 labels = ['lr=0.1', 'lr=0.01', 'lr=0.001', 'lr=0.0001']
----> 2 plot_horizontal_bar(labels, validation_accuracy_list, xlabel='Validation accurac

<ipython-input-15-d205da539b63> in plot_horizontal_bar(x, y, xlabel, ylabel, title, use_
      1 def plot_horizontal_bar(x, y, xlabel, ylabel, title, use_xlim=False):
----> 2     fig, ax = plt.subplots()
      3     width = 0.75 # the width of the bars
      4     ind = np.arange(len(y))  # the x locations for the groups
      5     ax.barh(ind, y, width, color="blue")

NameError: name 'plt' is not defined

SEARCH STACK OVERFLOW
```

```
validation_accuracy_list = []
```

Testing different batch size

- Using 3 dense layers with learning_rate = 0.001. Minibatches: batchsize=8, shuffle=True
- Using 3 dense layers with learning_rate = 0.001. Minibatches: batchsize=16, shuffle=True
- Using 3 dense layers with learning_rate = 0.001. Minibatches: batchsize=32, shuffle=True
- Using 3 dense layers with learning_rate = 0.001. Minibatches: batchsize=64, shuffle=True
- Using 3 dense layers with learning_rate = 0.001. Minibatches: batchsize=128, shuffle=True

```
dense_output_units=[100, 200, 300]

network = get_network(input_units=X_train.shape[1], output_units=10, epochs=25, dense_output_
```

Using 3 dense layers with learning_rate = 0.001. Minibatches: **batchsize=8**, shuffl

```
mlp_training(network, input_units=X_train.shape[1], output_units=10, learning_rate=0.001, epo
```

Using 3 dense layers with learning_rate = 0.001. Minibatches: **batchsize=16**, shuf

```
mlp_training(network, input_units=X_train.shape[1], output_units=10, learning_rate=0.001, epo
```

```
  0%|          | 0/2500 [00:00<?, ?it/s]
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-26-b38e3e9470f1> in <module>()
----> 1 mlp_training(network, input_units=X_train.shape[1], output_units=10, learning_ra

 2 frames
<ipython-input-10-4e87c32f38ed> in iterate_minibatches(inputs, targets, batchsize, shuff
      9         else:
     10             excerpt = slice(start_idx, start_idx + batchsize)
---> 11         yield inputs[excerpt], targets[excerpt]

TypeError: only integer scalar arrays can be converted to a scalar index
```

SEARCH STACK OVERFLOW

Using 3 dense layers with learning_rate = 0.001. Minibatches: **batchsize=32**, shuf

```
mlp_training(network, input_units=X_train.shape[1], output_units=10, learning_rate=0.001, epo
```

Using 3 dense layers with learning_rate = 0.001. Minibatches: **batchsize=64**, shuf

```
mlp_training(network, input_units=X_train.shape[1], output_units=10, learning_rate=0.001, epo
```

Using 3 dense layers with learning_rate = 0.001. Minibatches: **batchsize=128**, shu

```
mlp_training(network, input_units=X_train.shape[1], output_units=10, learning_rate=0.001, epo
```

Best batch size

- Highest validation accuracy (mean)

```
labels = ['batchsize=8', 'batchsize=16', 'batchsize=32', 'batchsize=64', 'batchsize=128']
plot_horizontal_bar(labels, validation_accuracy_list, xlabel='Validation accuracy', ylabel=''
```

Conclusion: Best MLP model

Using 3 dense layers with learning_rate = 0.001. Minibatches: batchsize=64, shuf

```
dense_output_units=[100, 200, 300]

network = get_network(input_units=X_train.shape[1], output_units=10, epochs=25, dense_output_
```

```
        NETWORK: Multi layer perceptron
                Dense(input_units=3072, output_units=100, learning_rate=0.1)
                ReLU()
                Dense(input_units=100, output_units=200, learning_rate=0.1)
                ReLU()
                Dense(input_units=200, output_units=300, learning_rate=0.1)
                ReLU()
                Dense(input_units=300, output_units=10, learning_rate=0.1)
```

```
mlp_training(network, input_units=X_train.shape[1], output_units=10, learning_rate=0.001, epo
```

```
  0%|          | 0/625 [00:00<?, ?it/s]
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-28-e6800a0bdee2> in <module>()
----> 1 mlp_training(network, input_units=X_train.shape[1], output_units=10, learning_ra

                                  2 frames

<ipython-input-10-4e87c32f38ed> in iterate_minibatches(inputs, targets, batchsize, shuff
      9         else:
     10             excerpt = slice(start_idx, start_idx + batchsize)
---> 11         yield inputs[excerpt], targets[excerpt]

TypeError: only integer scalar arrays can be converted to a scalar index

SEARCH STACK OVERFLOW
```