Lucerne University of
Applied Sciences and Arts

HOCHSCHULE
LUZERN

Master of Science (MSc)

Applied Information and Data Science

Institut für Natur- und Geisteswissenschaften ING

Prof. Dr. Mirko Birbaumer

Dozent

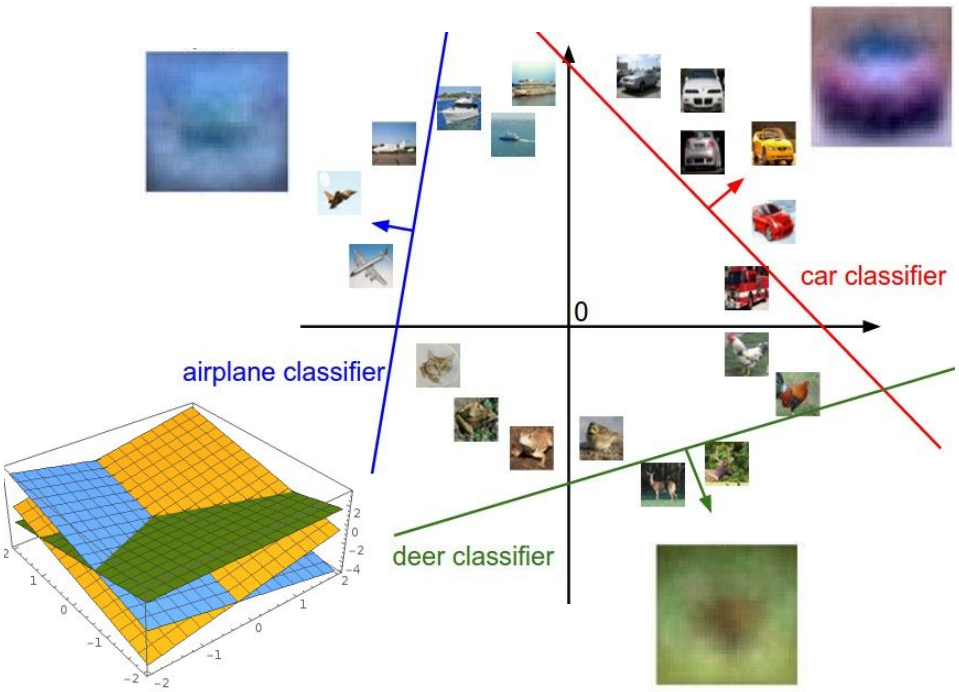
T direkt +41 41 349 33 40

mirko.birbaumer@hslu.ch

Luzern 30.09.2020

[Deep Learning in Vision]

Where we are now... : Linear Clasiffiers



Plot created using Wolfram Cloud

$$f(x,W) = Wx + b$$

Array of **32x32x3** numbers
(3072 numbers total)

Where we are now... : Cross-Entropy Loss Function

Softmax Classifier (Multinomial Logistic Regression)

$$L_i = -\log\left(\frac{e^{sy_i}}{\sum_j e^{s_j}}\right)$$

unnormalized probabilities

	unnormalized log probabilities	exp		normalize	probabilities	
cat	**3.2**	→	**24.5**	→	**0.13**	→ L_i = -log(0.13) = **0.89**
car	5.1	→	164.0	→	0.87	
frog	-1.7	→	0.18	→	0.00	

Where are we now... : Optimization

Recap

How do we find the best W?

- We have some dataset of (x,y)
- We have a **score function:** $s = f(x; W) \overset{\text{e.g.}}{=} Wx$
- We have a **loss function**:

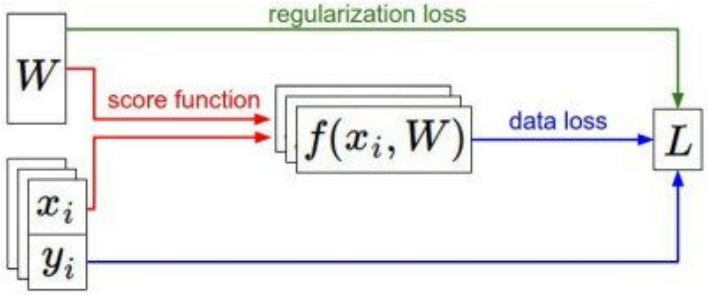
Softmax

$$L_i = -\log\left(\frac{e^{sy_i}}{\sum_j e^{s_j}}\right)$$

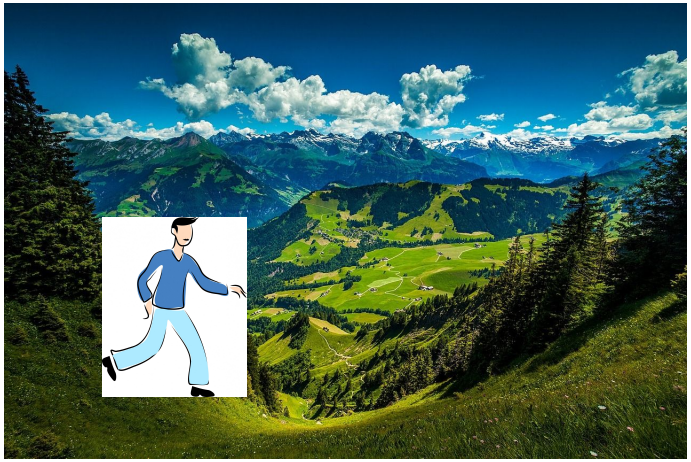
SVM

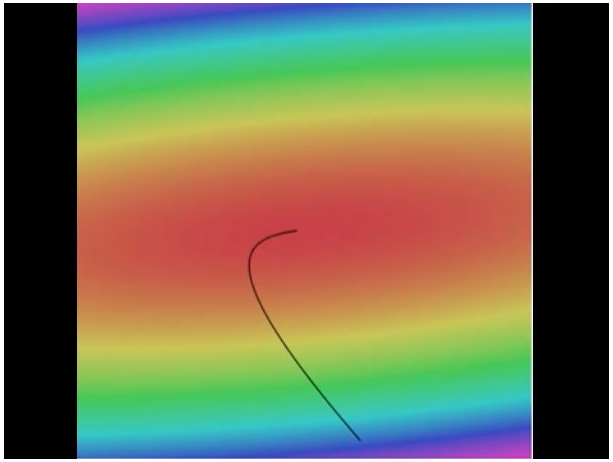
$$L_i = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + 1)$$

$$L = \frac{1}{N}\sum_{i=1}^{N} L_i + R(W)$$ Full loss



Where are we now… : Optimization





```
# Vanilla Gradient Descent

while True:
  weights_grad = evaluate_gradient(loss_fun, data, weights)
  weights += - step_size * weights_grad # perform parameter update
```

Where we are now … : Optimization

Mini-batch SGD

Loop:

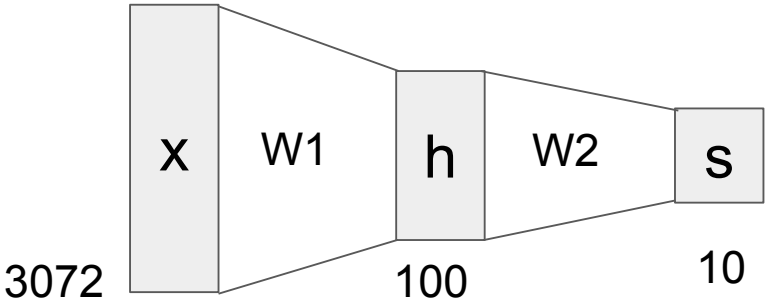
1. Sample a batch of data (32, 64, 128, …)
2. Forward prop it through the graph network, compute loss
3. Backprop to calculate the gradients
4. Update the parameters using the gradient

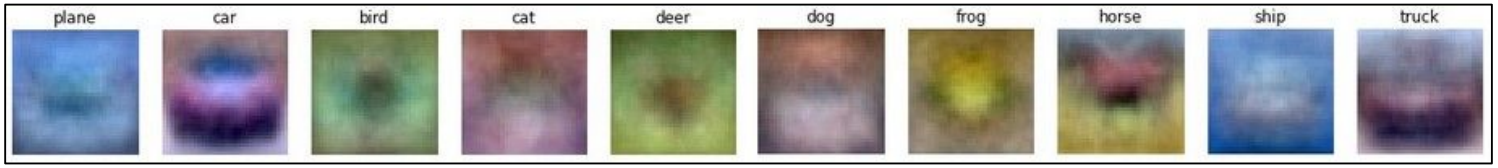
Where are we now... : Neural Networks

Neural networks: without the brain stuff

(**Before**) Linear score function: $f = Wx$

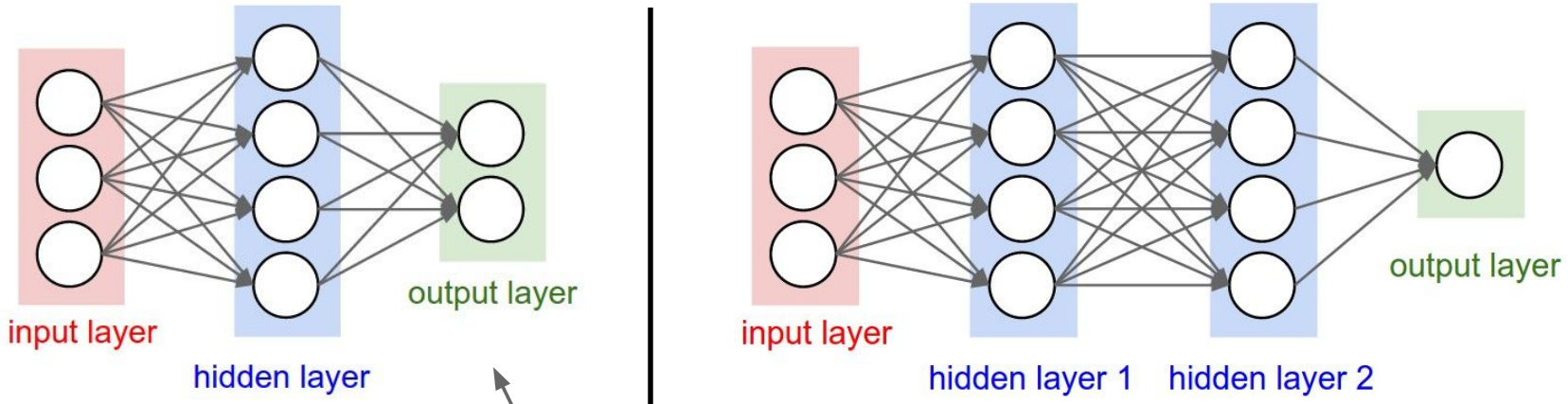
(**Now**) 2-layer Neural Network $f = W_2 \max(0, W_1 x)$





Where are we now … : Neural Architectures / see Live Voting

Neural networks: Architectures

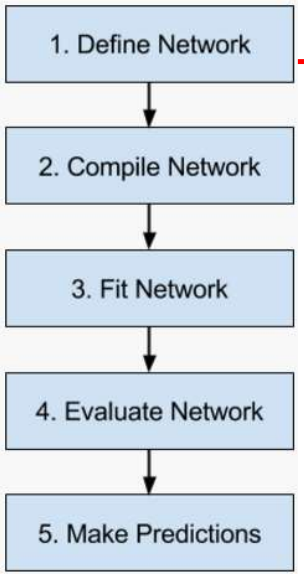


"2-layer Neural Net", or
"1-hidden-layer Neural Net"

"3-layer Neural Net", or
"2-hidden-layer Neural Net"

"Fully-connected" layers

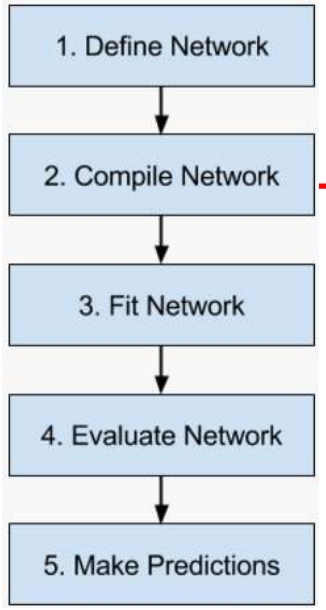
Neural Networks with Keras : Example MNIST



Number of neurons in (first) hidden dense layers (will be input to next layer)

```
model = tf.keras.Sequential()
# From Input to first hidden layer
model.add(tf.keras.layers.Dense(500, activation= tf.nn.sigmoid,
                                batch_input_shape=(None,784)))
model.add(tf.keras.layers.Dense(50,  activation=tf.nn.sigmoid))
model.add(tf.keras.layers.Dense(10,  activation=tf.nn.softmax))
```

Neural Networks with Keras : Example MNIST



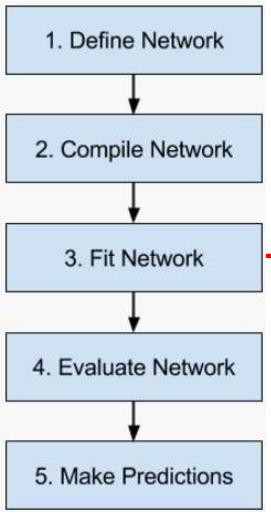
loss function that will be minimized

```
model.compile(loss='categorical_crossentropy',
              optimizer='adadelta',
              metrics=['accuracy'])
```

easiest optimizer is SGD
(stochastic gradient descent)

Which metrics besides 'loss' do
we want to collect during training

Neural Networks with Keras : Example MNIST



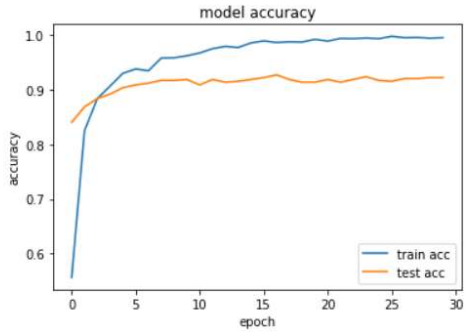

```
# define information required for tensorboard
tensorboard = keras.callbacks.TensorBoard(
    log_dir='tensorboard/mnist_small/' + name + '/',
    write_graph=True,
    histogram_freq=1)

# train the model, memorize training history
history = model.fit(X[0:2400],
          convertToOneHot(y[0:2400],10),
          epochs=30,
          batch_size=128,
          callbacks=[tensorboard],
          validation_data=[X[2400:3000],
                           convertToOneHot(y[2400:3000],10)]
```

input data (train)
output/label (train)

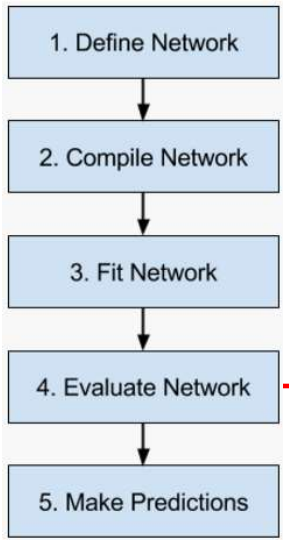
How and how often provide training data

In history we memorize development of loss and metrics achieved in successive training steps

```
plt.plot(history.history['acc'])
plt.plot(history.history['val_acc'])
plt.title('model accuracy')
plt.ylabel('accuracy')
plt.xlabel('epoch')
plt.legend(['train acc', 'test acc'], loc='lower right')
plt.show()
```



Neural Networks with Keras : Example MNIST




```
model.evaluate(X[0:2000], convertToOneHot(y[0:2000],10))
```

```
2000/2000 [==============================] - 0s 215us/step
```

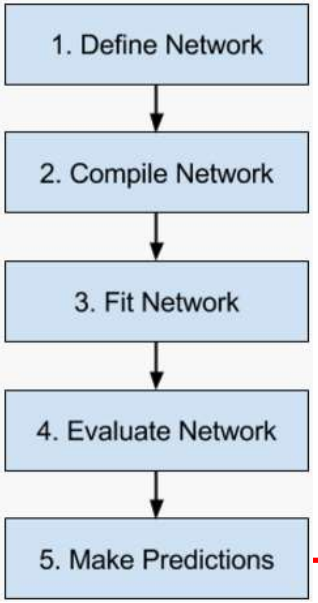
```
[2.5549037284851073, 0.1085]
```

loss

metrics: accuracy

```
model.compile(loss='categorical_crossentropy',
              optimizer='adadelta',
              metrics=['accuracy'])
```

Neural Networks with Keras : Example MNIST




```
model.predict_classes(X[0:10])
```

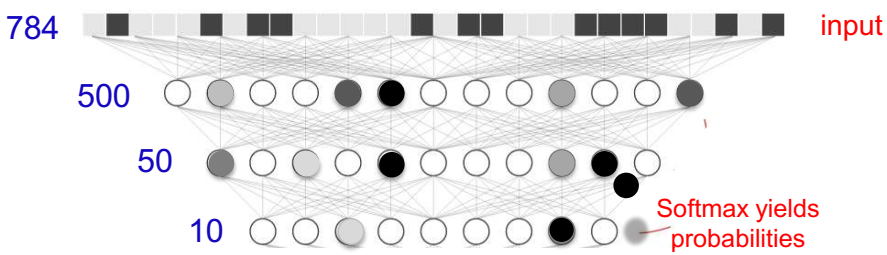
```
array([0, 9, 9, 1, 6, 9, 8, 9, 0, 0])
```

predicted class labels of the first 10 images

Neural Networks with Keras : Example MNIST

```
model.summary()
```

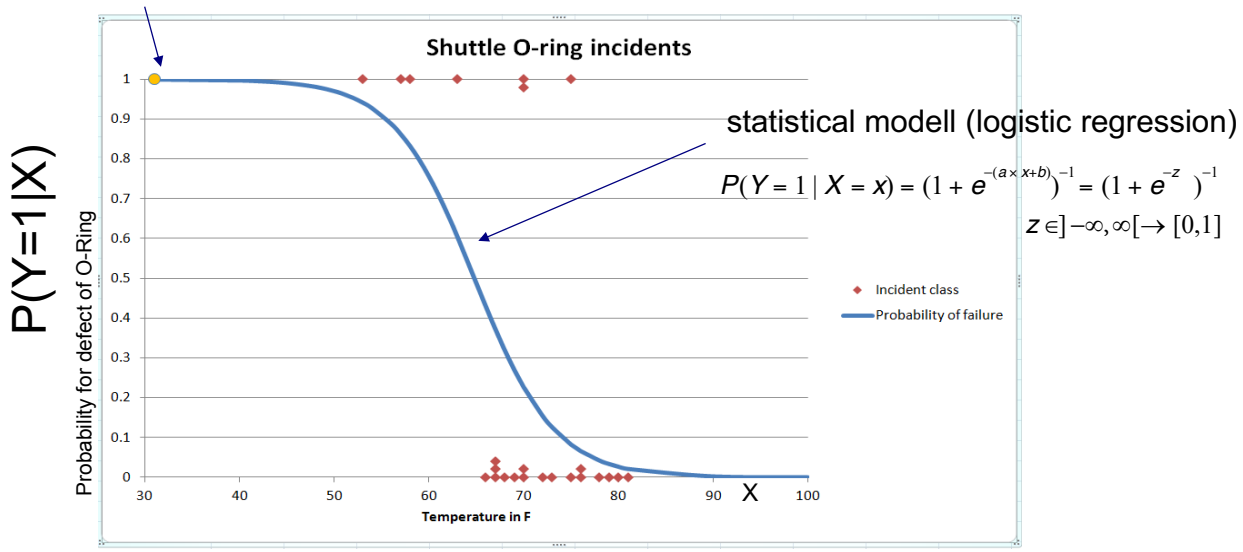
```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_1 (Dense)              (None, 500)               392500
_________________________________________________________________
activation_1 (Activation)    (None, 500)               0
_________________________________________________________________
dense_2 (Dense)              (None, 50)                25050
_________________________________________________________________
activation_2 (Activation)    (None, 50)                0
_________________________________________________________________
dense_3 (Dense)              (None, 10)                510
=================================================================
Total params: 418,060
Trainable params: 418,060
Non-trainable params: 0
_________________________________________________________________
```



Exercise 2.2 : Logistic Regression in 1D and Challenger O-Rings

Predict if O-Ring is broken, depending on temperature

Challenger launch @31 F
Prob. of a failure=0.9997



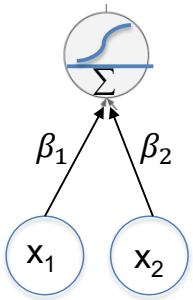
statistical modell (logistic regression)

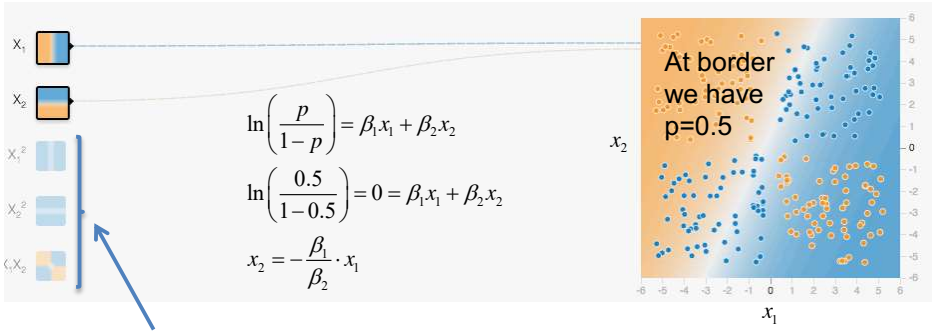
$$P(Y=1 \mid X=x) = (1+e^{-(a\times x+b)})^{-1} = (1+e^{-z})^{-1}$$

$$z \in]-\infty,\infty[\rightarrow [0,1]$$

Exercise 2.2 : Logistic Regression in 2D

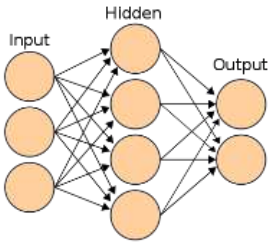
NN **without** hidden layer and sigmoid activation function yields linear separation curve.

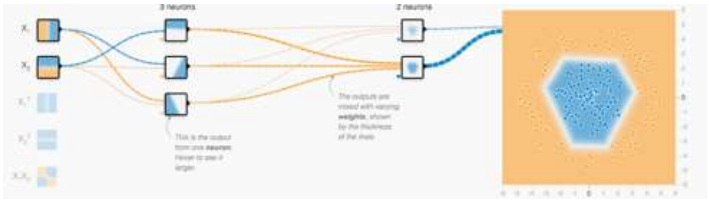




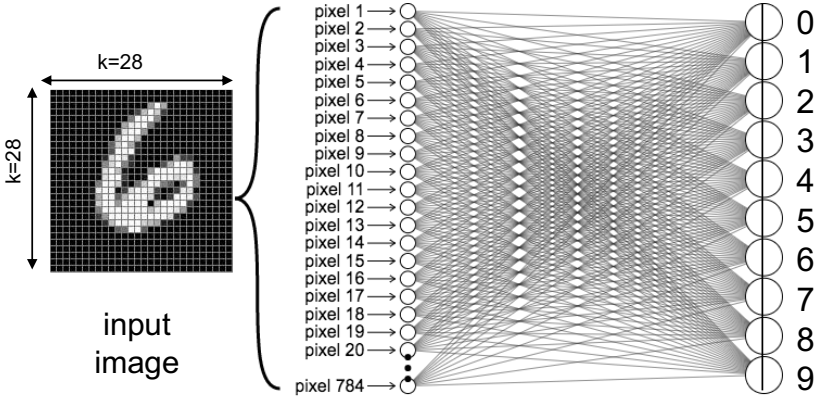
"handcrafted feature"

NN with ≥1 hidden layer and sigmoid activation function yields arbitrary separation curve.





Exercise 2.3 : Multinomial Logistic Regression and MNIST



$$z_j = \sum_i \left(x_i \cdot w_{ij} \right)$$

$$p_j = \frac{e^{z_j}}{\sum_{k=0}^{9} e^{z_k}}$$

Each output node z_j gets 784 inputs connected via 784 weights which (usually the weights are different for each node) can also be arranged as 28x28 weight image

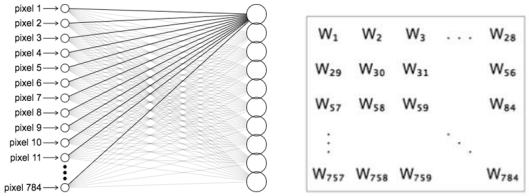
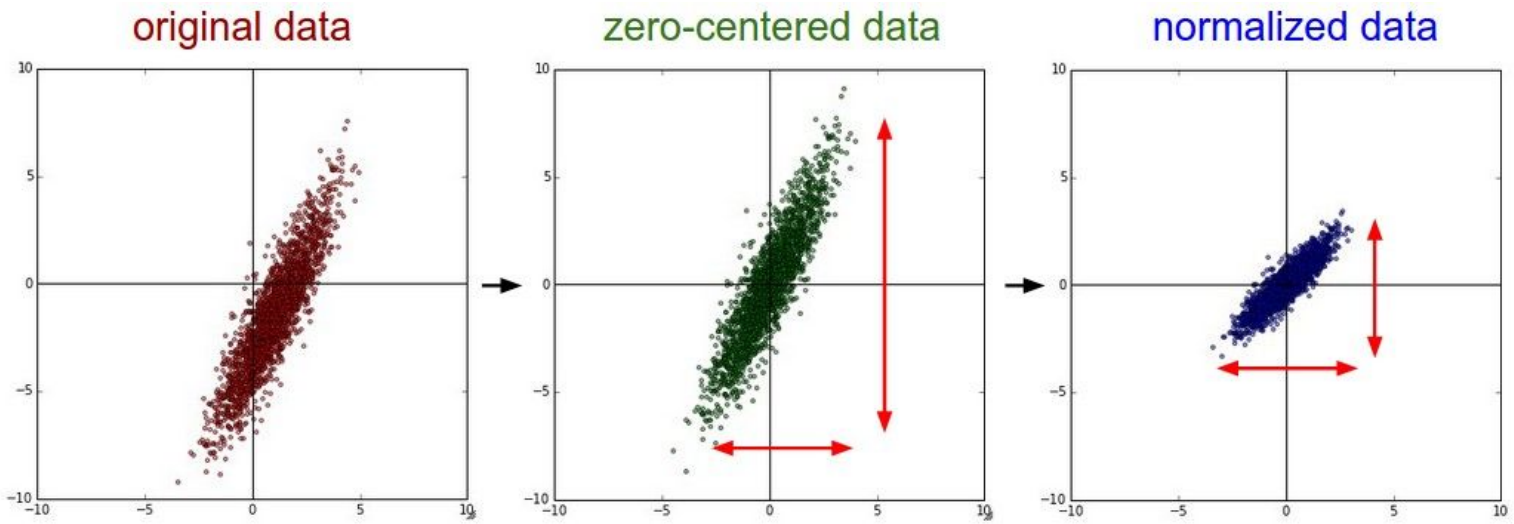


Image credits: https://ml4a.github.io/ml4a/looking_inside_neural_nets/

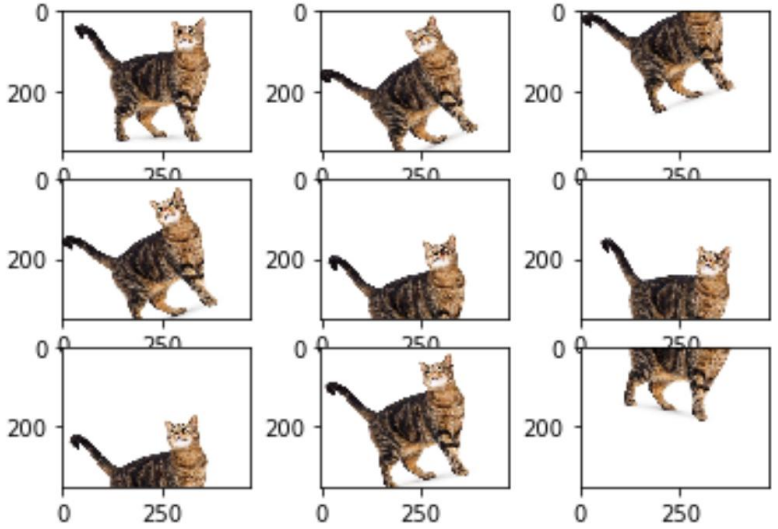
Data Preprocessing : Mean Subtraction and Normalization / see Jupyter Notebook Part 1.1



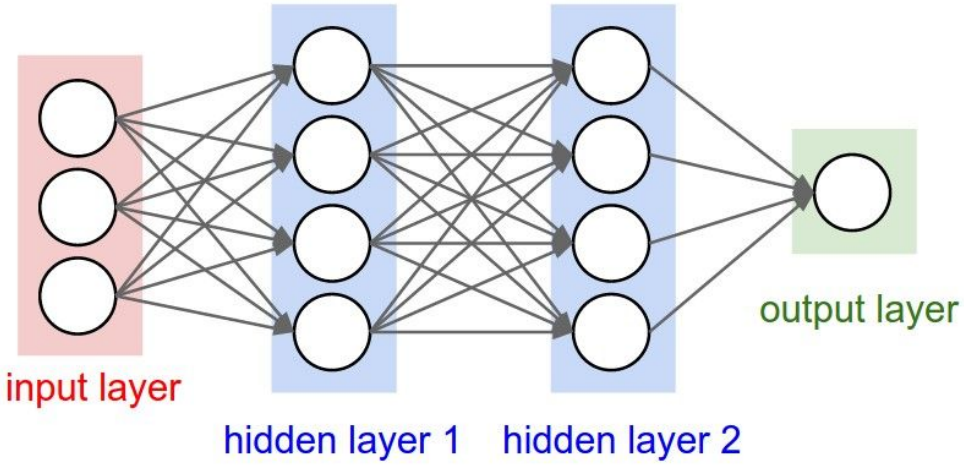
Data Preprocessing in Practice for Images: Center only

- Consider CIFAR-10 example with [32,32,3] images
- Subtract the mean image (e.g. AlexNet) : (mean image = [32,32,3] array)
- Subtract per-channel mean (e.g. VGGNet) : (mean along each channel = 3 numbers)
- Not common to normalize variance, to do PCA or whitening

Data Augmentation / see Jupyter Notebook Part 1.2



Choose the Architecture / see Jupyter Notebook Part 2

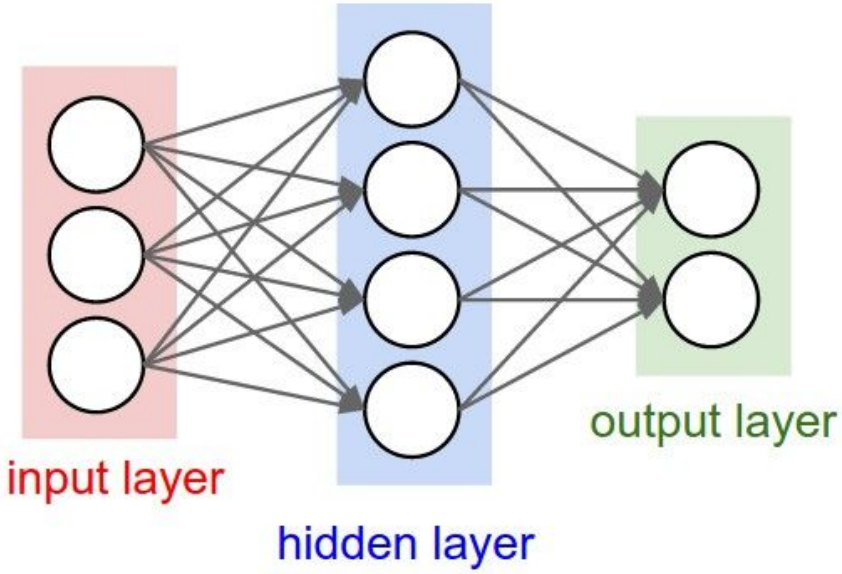


"3-layer Neural Net", or
"2-hidden-layer Neural Net"

Say we start with 2 hidden layers : 512 units and 128 units, resp., and output layer with 10 units for CIFAR-10

Weight Initialization

- Q: what happens when W=0 init is used?



Weight Initialization / see Jupyter Notebook Part 4

- First idea: **Small random numbers**

(gaussian with zero mean and 1e-2 standard deviation)

```
W = 0.01* np.random.randn(D,H)
```

Works ~okay for small networks, but problems with deeper networks.

Batch Normalization [Ioffe and Szegedy, 2015]

"you want unit gaussian activations? just make them so."

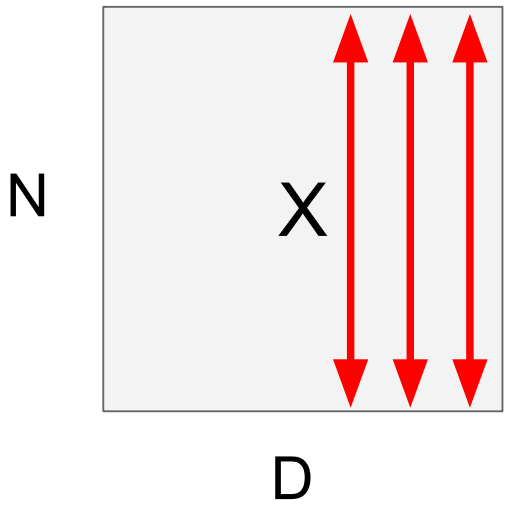
consider a batch of activations at some layer.
To make each dimension unit gaussian, apply:

$$\widehat{x}^{(k)} = \frac{x^{(k)} - \mathrm{E}[x^{(k)}]}{\sqrt{\mathrm{Var}[x^{(k)}]}}$$

this is a vanilla differentiable function...

Batch Normalization [Ioffe and Szegedy, 2015]

"you want unit gaussian activations?
just make them so."

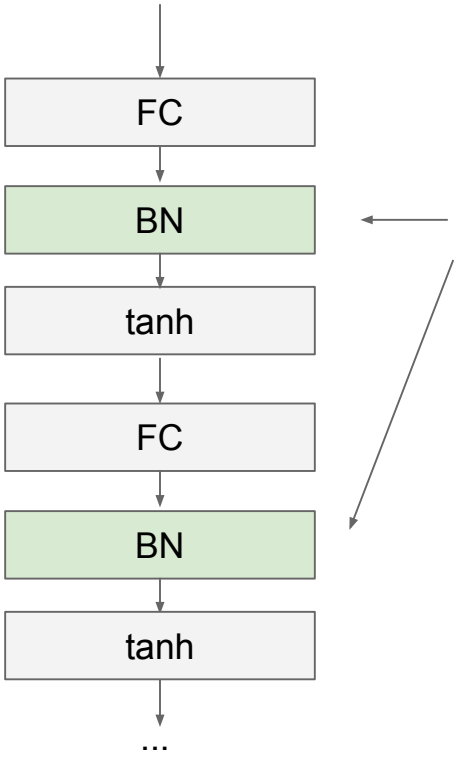


1. compute the empirical mean and variance independently for each dimension.

2. Normalize

$$\widehat{x}^{(k)} = \frac{x^{(k)} - \mathrm{E}[x^{(k)}]}{\sqrt{\mathrm{Var}[x^{(k)}]}}$$

Batch Normalization [Ioffe and Szegedy, 2015]



Usually inserted after Fully Connected or Convolutional layers, and before nonlinearity.

$$\widehat{x}^{(k)} = \frac{x^{(k)} - \mathrm{E}[x^{(k)}]}{\sqrt{\mathrm{Var}[x^{(k)}]}}$$

Batch Normalization [Ioffe and Szegedy, 2015]

Normalize:

$$\widehat{x}^{(k)} = \frac{x^{(k)} - \mathrm{E}[x^{(k)}]}{\sqrt{\mathrm{Var}[x^{(k)}]}}$$

And then allow the network to squash the range if it wants to:

$$y^{(k)} = \gamma^{(k)}\widehat{x}^{(k)} + \beta^{(k)}$$

Note, the network can learn:

$$\gamma^{(k)} = \sqrt{\mathrm{Var}[x^{(k)}]}$$

$$\beta^{(k)} = \mathrm{E}[x^{(k)}]$$

to recover the identity mapping.

Batch Normalization / see Jupyter Notebook Part 5

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β

Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma^2_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma\widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

Note: at test time BatchNorm layer functions differently:

The mean/std are not computed based on the batch. Instead, a single fixed empirical mean of activations during training is used.

(e.g. can be estimated during training with running averages)

Batch Normalization : Advantages

Nice properties of Batch Normalization:

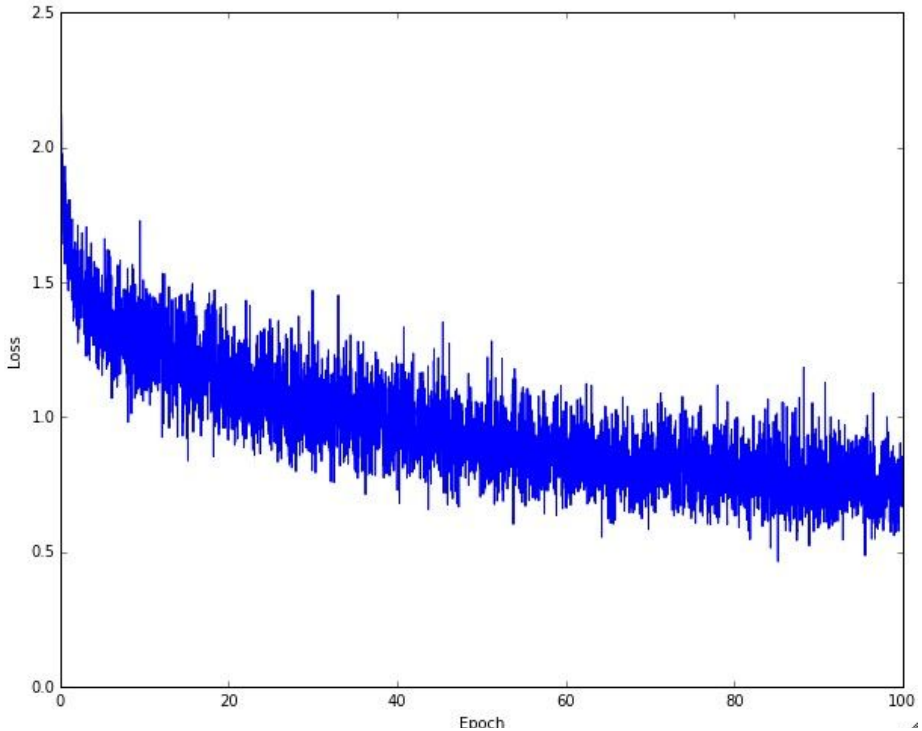
1. Improves gradient flow through the network
2. Allows higher learning rates
3. Reduces strong dependence on initialization
4. Acts as a form of regularization in a funny way, and slightly reduces the need for dropout, maybe

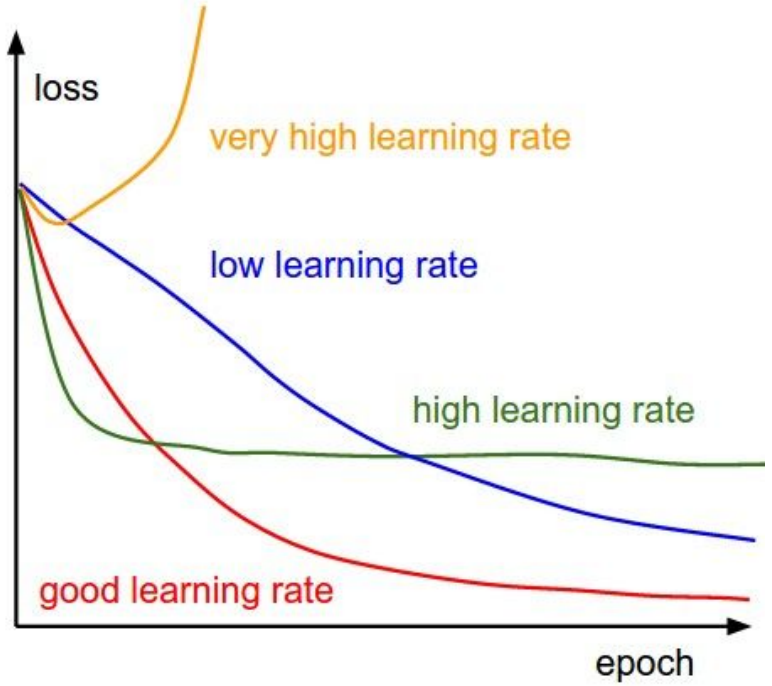
Double check that loss is reasonable / see Jupyter Notebook Part 6

- **Look for correct loss at chance performance** : For example, for CIFAR-10 with a Softmax classifier we would expect the initial loss to be 2.302, because we expect a diffuse probability of 0.1 for each class (since there are 10 classes), and Softmax loss is the negative log probability of the correct class so: -ln(0.1) = 2.302

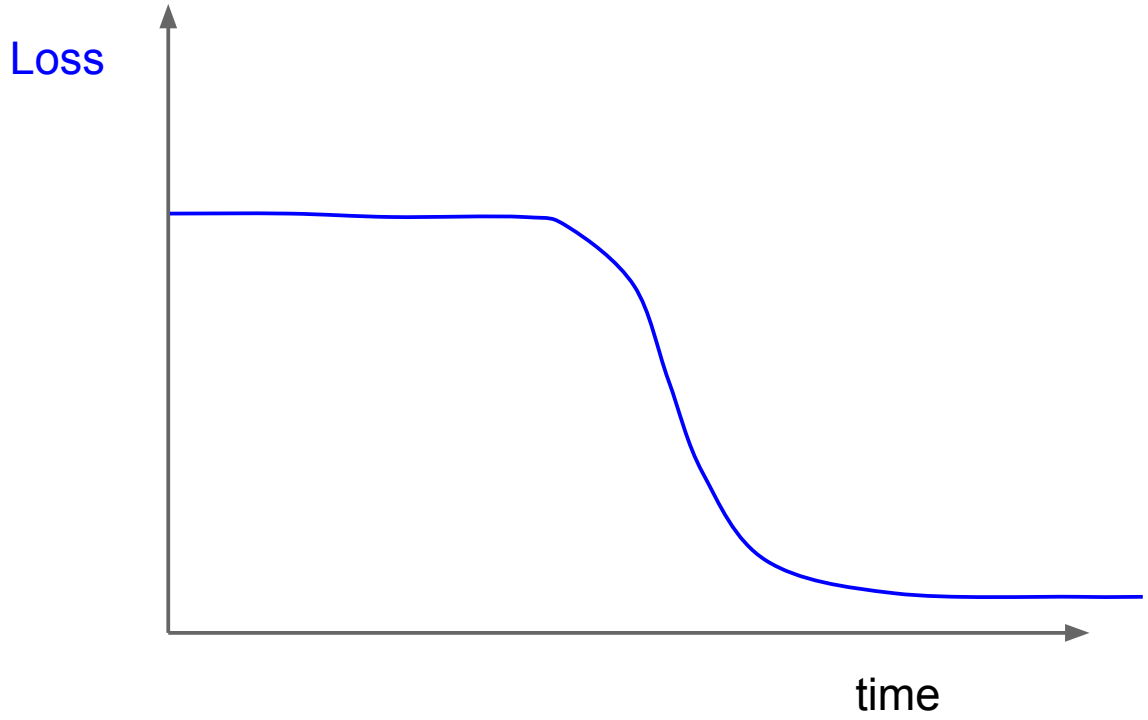
- **Overfit small data set** : before training on the full dataset try to train on a tiny portion (e.g. 20 examples) of your data and make sure you can achieve zero cost. For this experiment it is also best to set regularization to zero, otherwise this can prevent you from getting zero cost.

Monitor and Visualize the Loss Curve

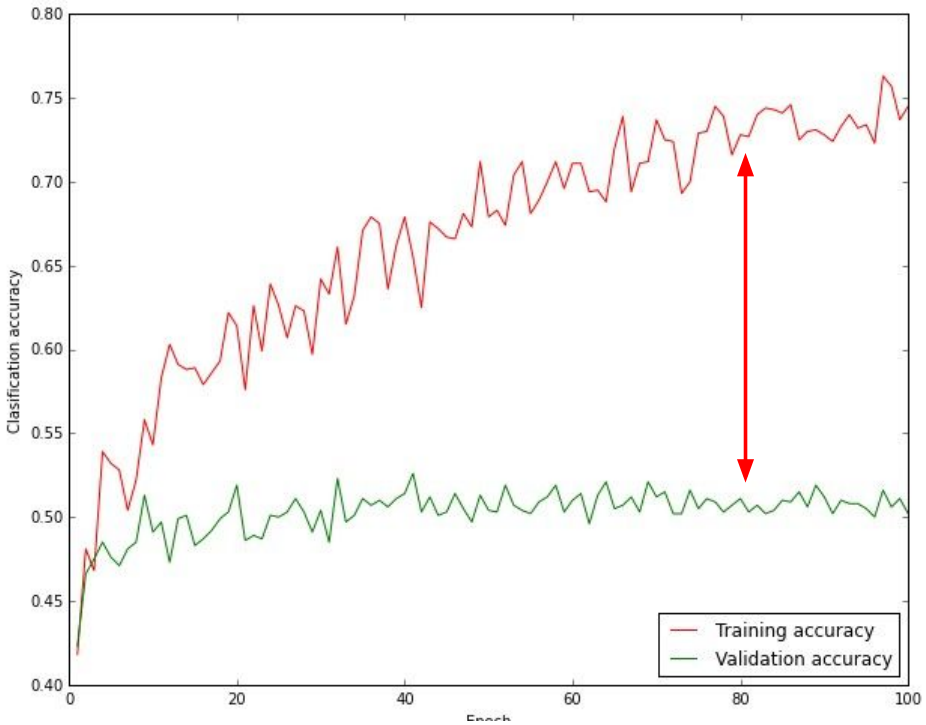




Monitor and Visualize the Loss Curve : Bad Initialization



Monitor and Visualize the Accuracy



big gap = overfitting

=> increase regularization strength?

no gap

=> increase model capacity?

Regularization / see Jupyter Notebook Part 7

$$L = \frac{1}{N}\sum_{i=1}^{N}\sum_{j\neq y_i}\max(0, f(x_i; W)_j - f(x_i; W)_{y_i} + 1) + \boxed{\lambda R(W)}$$

In common use:

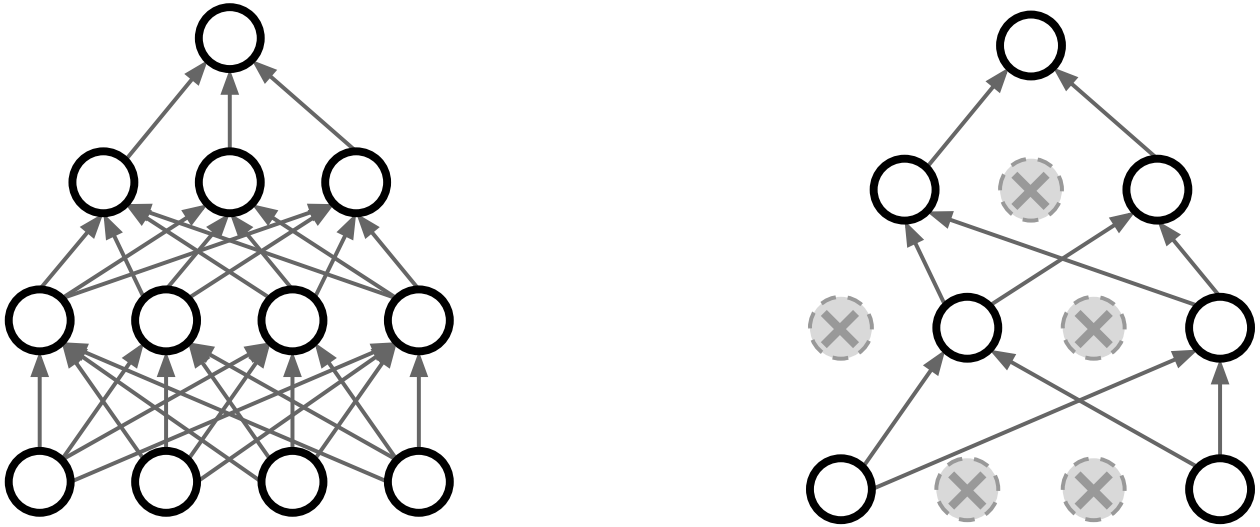
L2 regularization $R(W) = \sum_k \sum_l W_{k,l}^2$ (Weight decay)

L1 regularization $R(W) = \sum_k \sum_l |W_{k,l}|$

Elastic net (L1 + L2) $R(W) = \sum_k \sum_l \beta W_{k,l}^2 + |W_{k,l}|$

Regularization : Dropout

In each forward pass, randomly set some neurons to zero
Probability of dropping is a hyperparameter; 0.5 is common

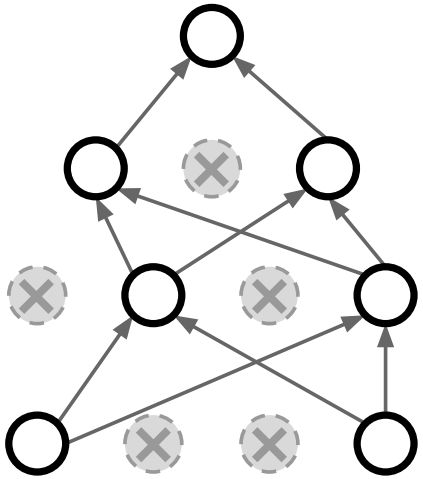


Srivastava et al, “Dropout: A simple way to prevent neural networks from overfitting”, JMLR 2014

Regularization : Dropout

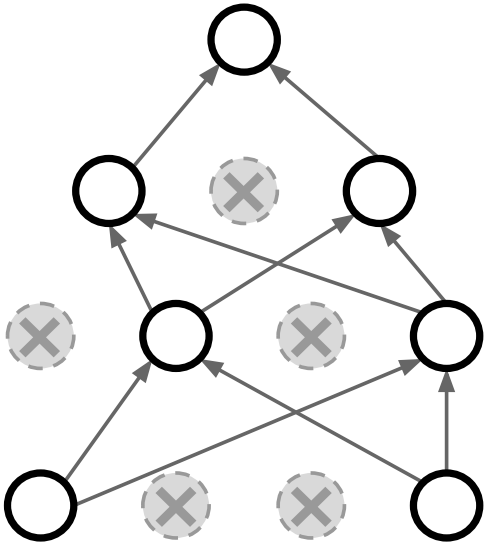
```python
p = 0.5 # probability of keeping a unit active. higher = less dropout

def train_step(X):
  """ X contains the data """

  # forward pass for example 3-layer neural network
  H1 = np.maximum(0, np.dot(W1, X) + b1)
  U1 = np.random.rand(*H1.shape) < p # first dropout mask
  H1 *= U1 # drop!
  H2 = np.maximum(0, np.dot(W2, H1) + b2)
  U2 = np.random.rand(*H2.shape) < p # second dropout mask
  H2 *= U2 # drop!
  out = np.dot(W3, H2) + b3

  # backward pass: compute gradients... (not shown)
  # perform parameter update... (not shown)
```

Example forward pass with a 3-layer network using dropout



Regularization : Dropout

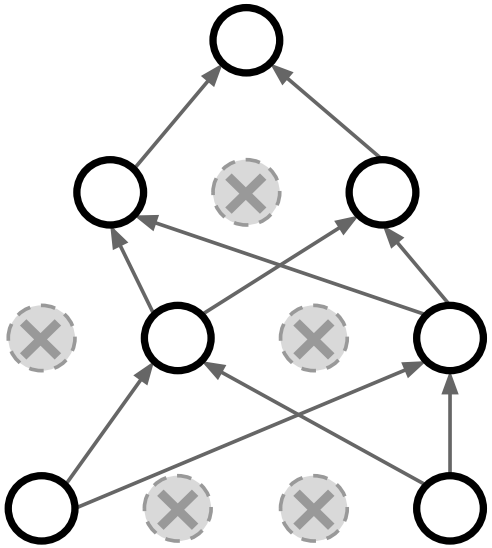
How can this possibly be a good idea?



Forces the network to have a redundant representation;
Prevents co-adaptation of features

Regularization : Dropout / see Jupyter Notebook Part 7.3

How can this possibly be a good idea?



Another interpretation:

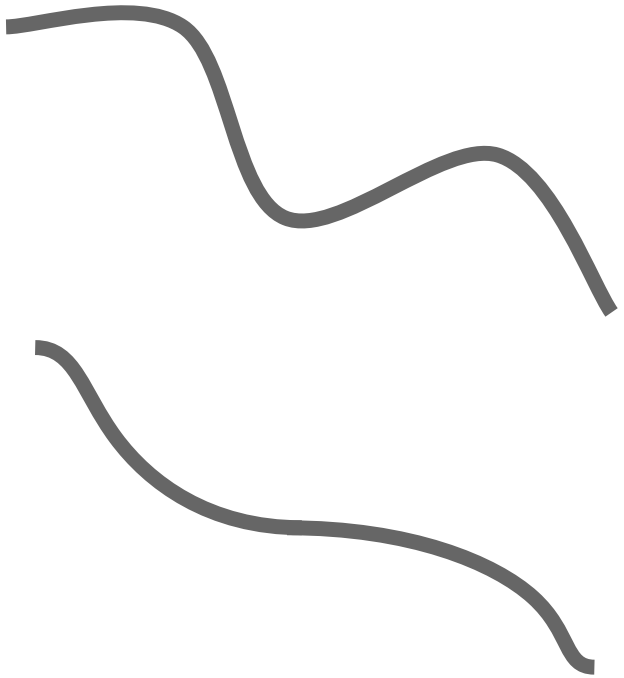
Dropout is training a large **ensemble** of models (that share parameters).

Each binary mask is one model

An FC layer with 4096 units has
$2^{4096} \sim 10^{1233}$ possible masks!
Only $\sim 10^{82}$ atoms in the universe...

Optimization : Problems with SGD

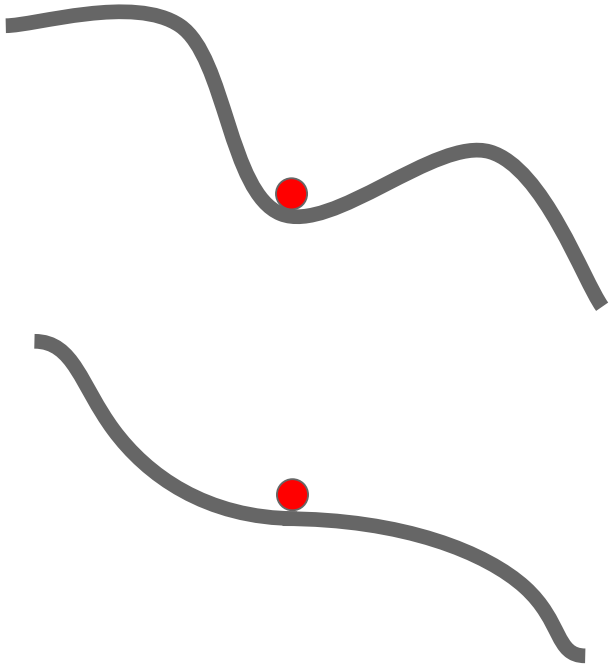
What if the loss function has a **local minima** or **saddle point**?



Optimization : Problems with SGD

What if the loss function has a **local minima** or **saddle point**?

Zero gradient, gradient descent gets stuck



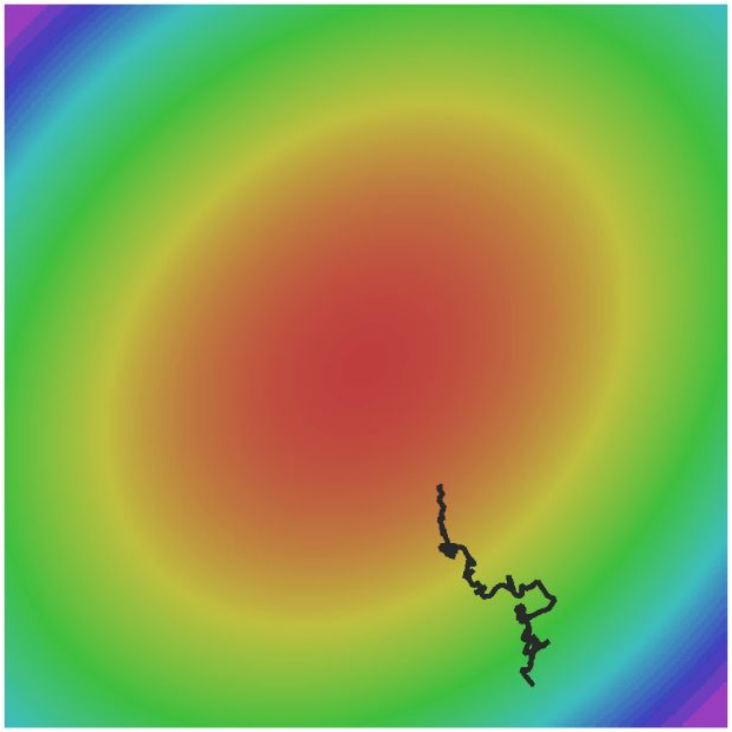
Optimization : Problems with SGD

-

Our gradients come from minibatches so they can be noisy!

$$L(W) = \frac{1}{N} \sum_{i=1}^{N} L_i(x_i, y_i, W)$$

$$\nabla_W L(W) = \frac{1}{N} \sum_{i=1}^{N} \nabla_W L_i(x_i, y_i, W)$$



Optimization : Problems with SGD

SGD

$$x_{t+1} = x_t - \alpha \nabla f(x_t)$$

```
while True:
  dx = compute_gradient(x)
  x += learning_rate * dx
```

SGD+Momentum

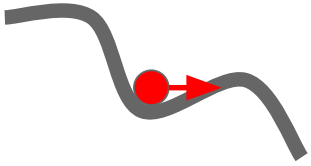
$$v_{t+1} = \rho v_t + \nabla f(x_t)$$
$$x_{t+1} = x_t - \alpha v_{t+1}$$

```
vx = 0
while True:
  dx = compute_gradient(x)
  vx = rho * vx + dx
  x += learning_rate * vx
```

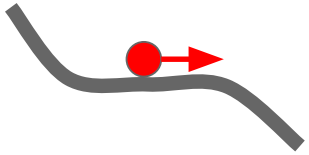
- Build up “velocity” as a running mean of gradients
- Rho gives “friction”; typically rho=0.9 or 0.99

Optimization : Problems with SGD

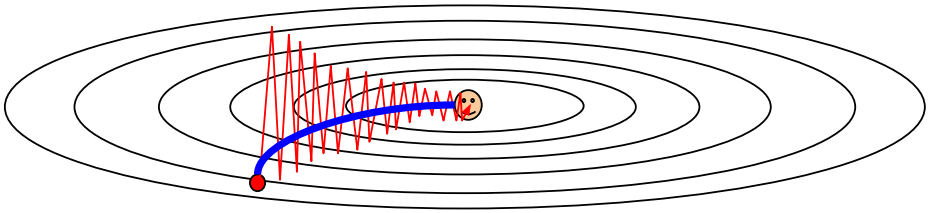
Local Minima



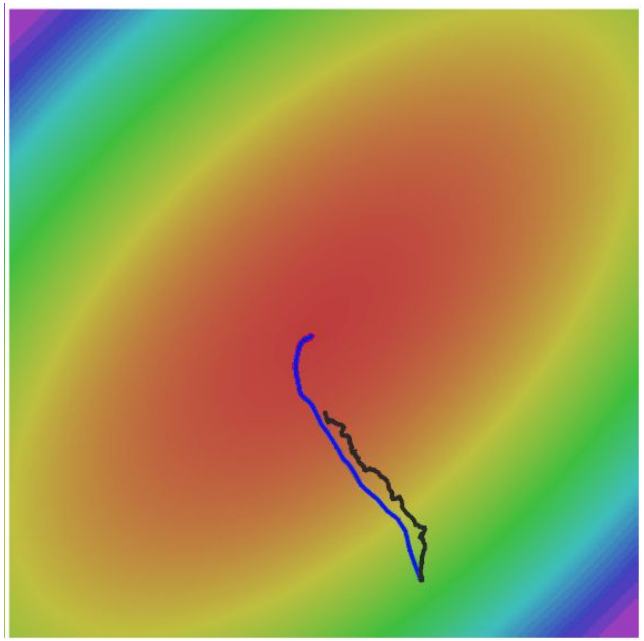
Saddle points



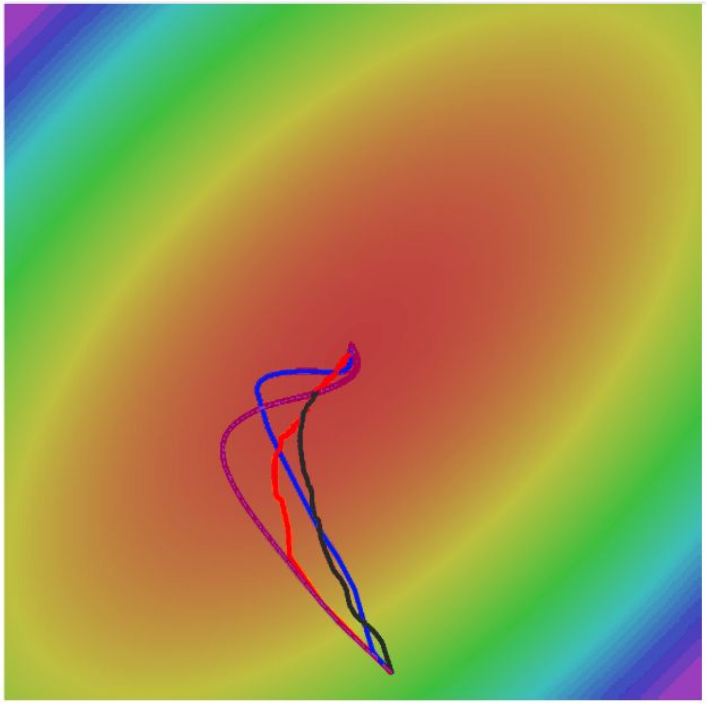
Poor Conditioning



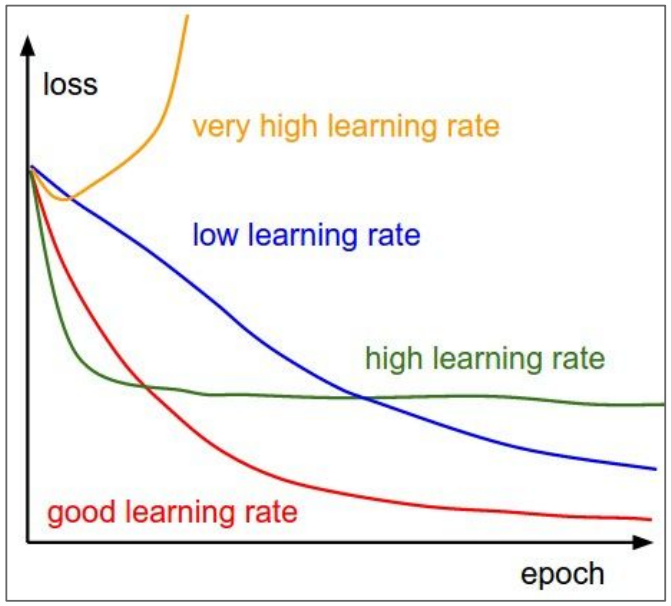
Gradient Noise



Adam / see Jupyter Notebook

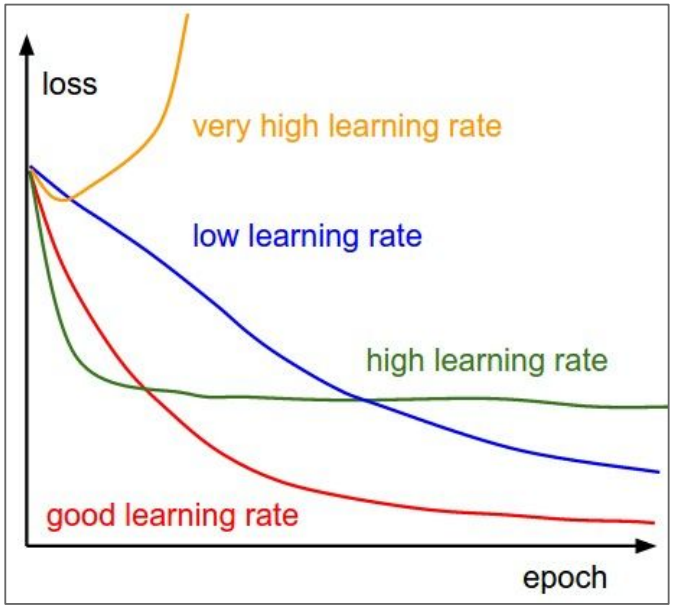


SGD, SGD+Momentum, Adagrad, RMSProp, Adam all have learning rate as a hyperparameter



Q: Which one of these learning rates is best to use?

SGD, SGD+Momentum, Adagrad, RMSProp, Adam all have learning rate as a hyperparameter.



=> Learning rate decay over time!

step decay:
e.g. decay learning rate by half every few epochs.

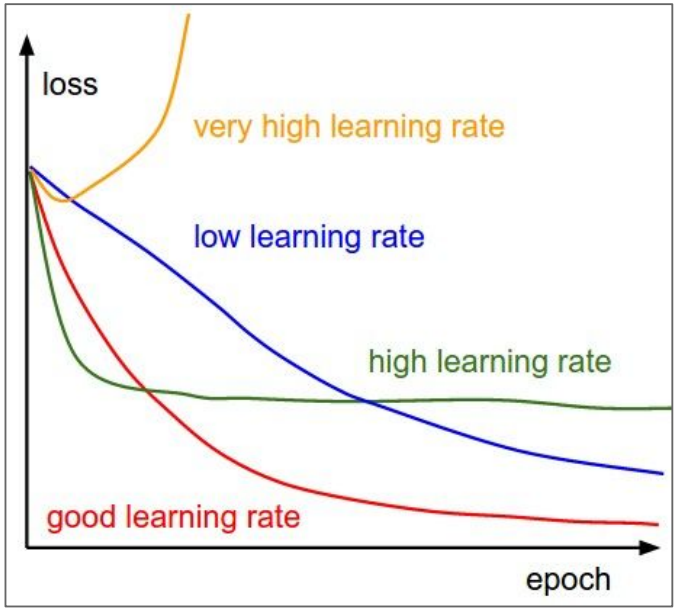
exponential decay:

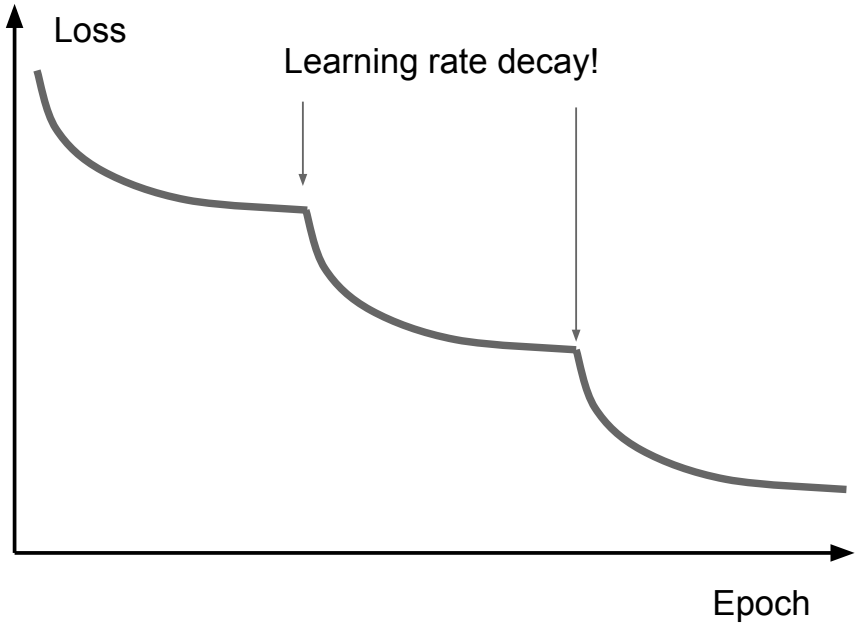
$$\alpha = \alpha_0 e^{-kt}$$

1/t decay:

$$\alpha = \alpha_0 / (1 + kt)$$

SGD, SGD+Momentum, Adagrad, RMSProp, Adam all have learning rate as a hyperparameter.





Hyperparameter Optimization

Hyperparameters to play with:

- network architecture
- learning rate, its decay schedule, update type
- regularization (L2/Dropout strength)

neural networks practitioner
music = loss function

Hyperparameter Optimization : Cross-validation strategy

- coarse -> fine cross-validation in stages
- First stage: only a few epochs to get rough idea of what params work
- Second stage: longer running time, finer search
  … (repeat as necessary)

For example: run coarse search for 5 epochs

```
max_count = 100
for count in xrange(max_count):
      reg = 10**uniform(-5, 5)
      lr = 10**uniform(-3, -6)

      trainer = ClassifierTrainer()
      model = init_two_layer_model(32*32*3, 50, 10) # input size, hidden size, number of classes
      trainer = ClassifierTrainer()
      best_model_local, stats = trainer.train(X_train, y_train, X_val, y_val,
                                              model, two_layer_net,
                                              num_epochs=5, reg=reg,
                                              update='momentum', learning_rate_decay=0.9,
                                              sample_batches = True, batch_size = 100,
                                              learning_rate=lr, verbose=False)
```

note it's best to optimize in log space!

```
val_acc: 0.412000, lr: 1.405206e-04, reg: 4.793564e-01, (1 / 100)
val_acc: 0.214000, lr: 7.231888e-06, reg: 2.321281e-04, (2 / 100)
val_acc: 0.208000, lr: 2.119571e-06, reg: 8.011857e+01, (3 / 100)
val_acc: 0.196000, lr: 1.551131e-05, reg: 4.374936e-05, (4 / 100)
val_acc: 0.079000, lr: 1.753300e-05, reg: 1.200424e+03, (5 / 100)
val_acc: 0.223000, lr: 4.215128e-05, reg: 4.196174e+01, (6 / 100)
val_acc: 0.441000, lr: 1.750259e-04, reg: 2.110807e-04, (7 / 100)
val_acc: 0.241000, lr: 6.749231e-05, reg: 4.226413e+01, (8 / 100)
val_acc: 0.482000, lr: 4.296863e-04, reg: 6.642555e-01, (9 / 100)
val_acc: 0.079000, lr: 5.401602e-06, reg: 1.599828e+04, (10 / 100)
val_acc: 0.154000, lr: 1.618508e-06, reg: 4.925252e-01, (11 / 100)
```

nice

Now run finer search...

```
max_count = 100
for count in xrange(max_count):
      reg = 10**uniform(-5, 5)
      lr = 10**uniform(-3, -6)
```

adjust range

```
max_count = 100
for count in xrange(max_count):
      reg = 10**uniform(-4, 0)
      lr = 10**uniform(-3, -4)
```

```
val_acc: 0.527000, lr: 5.340517e-04, reg: 4.097824e-01, (0 / 100)
val_acc: 0.492000, lr: 2.279484e-04, reg: 9.991345e-04, (1 / 100)
val_acc: 0.512000, lr: 8.680827e-04, reg: 1.349727e-02, (2 / 100)
val_acc: 0.461000, lr: 1.028377e-04, reg: 1.220193e-02, (3 / 100)
val_acc: 0.460000, lr: 1.113730e-04, reg: 5.244309e-02, (4 / 100)
val_acc: 0.498000, lr: 9.477776e-04, reg: 2.001293e-03, (5 / 100)
val_acc: 0.469000, lr: 1.484369e-04, reg: 4.328313e-01, (6 / 100)
val_acc: 0.522000, lr: 5.586261e-04, reg: 2.312685e-04, (7 / 100)
val_acc: 0.530000, lr: 5.808183e-04, reg: 8.259964e-02, (8 / 100)
val_acc: 0.489000, lr: 1.979168e-04, reg: 1.010889e-04, (9 / 100)
val_acc: 0.490000, lr: 2.036031e-04, reg: 2.406271e-03, (10 / 100)
val_acc: 0.475000, lr: 2.021162e-04, reg: 2.287807e-01, (11 / 100)
val_acc: 0.460000, lr: 1.135527e-04, reg: 3.905040e-02, (12 / 100)
val_acc: 0.515000, lr: 6.947668e-04, reg: 1.562808e-02, (13 / 100)
val_acc: 0.531000, lr: 9.471549e-04, reg: 1.433895e-03, (14 / 100)
val_acc: 0.509000, lr: 3.140888e-04, reg: 2.857518e-01, (15 / 100)
val_acc: 0.514000, lr: 6.438349e-04, reg: 3.033781e-01, (16 / 100)
val_acc: 0.502000, lr: 3.921784e-04, reg: 2.707126e-04, (17 / 100)
val_acc: 0.509000, lr: 9.752279e-04, reg: 2.850865e-03, (18 / 100)
val_acc: 0.500000, lr: 2.412048e-04, reg: 4.997821e-04, (19 / 100)
val_acc: 0.466000, lr: 1.319314e-04, reg: 1.189915e-02, (20 / 100)
val_acc: 0.516000, lr: 8.039527e-04, reg: 1.528291e-02, (21 / 100)
```

53% - relatively good for a 2-layer neural net with 50 hidden neurons.

Hyperparameter Optimization

Random Search vs. Grid Search

Random Search for Hyper-Parameter Optimization
Bergstra and Bengio, 2012

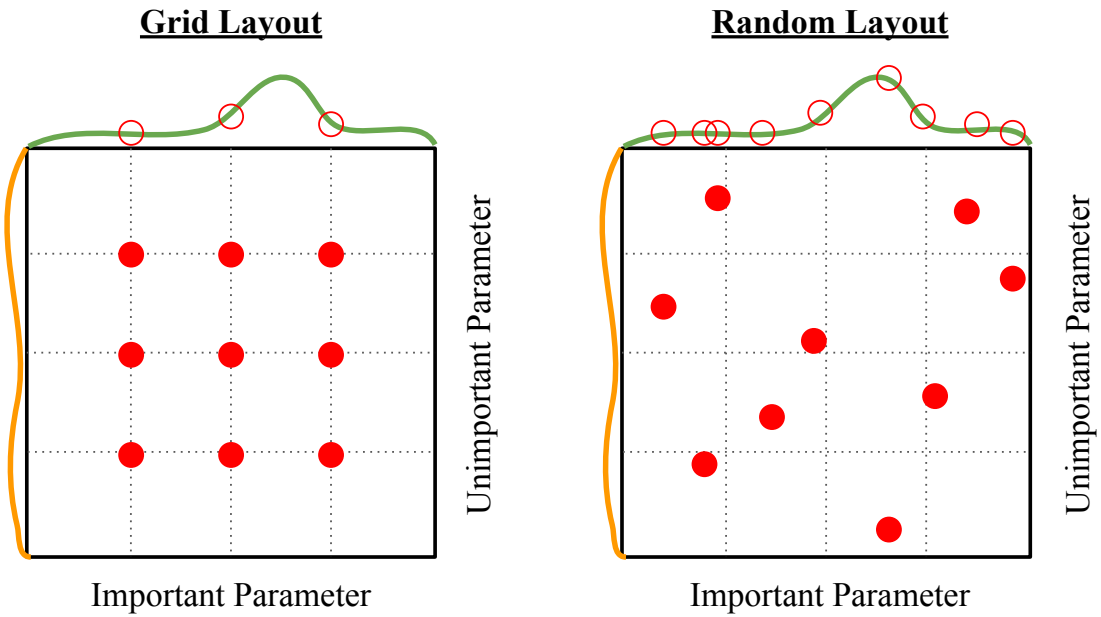


Illustration of Bergstra et al., 2012 by Shayne Longpre, copyright CS231n 2017

Summary

- Activation Functions (use ReLU)
- Data Preprocessing (images: subtract mean)
- Weight Initialization (use Xavier init)
- Batch Normalization (use)
- Babysitting the Learning process
- Parameter Update (use Adam)
- Decaying learning rate (use)
- Hyperparameter optimization (random sample hyperparams, in log space)

Tensorboard Basics / see Jupyter Notebook Part 10