Training set: observations of wealth and religiousness features with #blue or #red labels
Test set: How well can we do?
K-Nearest Neighbors (KNN) defines neighborhoods
K-Nearest Neighbors (KNN) $k=1$ classification results
K-Nearest Neighbors (KNN) $k=3$ classification results
K-Nearest Neighbors (KNN) $k=29$ classification results
A straight line can classify three points arbitrarily labeled
A straight line can not classify four points arbitrarily labeled
The capacity (Vapnik-Chervonenkis dimension) of a model describes how many points can be correctly predicted when they are produced by an adversary.
The capacity of non-parametric models is defined by the size of their training set

- k-nearest neighbor (KNN) regression computes its output based upon the k “nearest” training examples
- Often the best method, and certainly a baseline to beat
The **generalizability** of a model describes its ability to perform well on previously unseen inputs.
ML: Define $\mathcal{H}$, obtain $\hat{h}$

$\mathcal{H} \subset \mathcal{F}$, where $\mathcal{F}$ is the set of all functions mapping $\mathcal{X}$ onto $\mathcal{Y}$.

**Goal:** find $\hat{h} \in \mathcal{H}$

**Step 1:** define suitable hypothesis space $\mathcal{H}$

**Problem Set 1**

$x \in [0, 1]^{784}$

$\hat{y} \in [0, 1]^{10}$

$\mathbf{W} \in \mathbb{R}^{784 \times 10}$

$b \in \mathbb{R}^{10}$

$\phi_{\text{softmax}}(z)_i = \frac{e^{z_i}}{\sum_{j=1}^{\mid z \mid} e^{z_j}}$

$h(x; \mathbf{W}, b) = \phi_{\text{softmax}}(\mathbf{W}^\top x + b)$

$\mathcal{H} = \{ h(x; \mathbf{W}, b) | \mathbf{W} \in \mathbb{R}^{784 \times 10}, b \in \mathbb{R}^{10} \}$
ML: Define $\mathcal{H}$, obtain $\hat{h}$

$\mathcal{H} \subset \mathcal{F}$, where $\mathcal{F}$ is the set of all functions mapping $\mathcal{X}$ onto $\mathcal{Y}$.

**Goal:** find $\hat{h} \in \mathcal{H}$

**Step 1:** define suitable hypothesis space $\mathcal{H}$

**Problem Set 1**

- $\mathbf{x} \in [0, 1]^{784}$
- $\hat{\mathbf{y}} \in [0, 1]^{10}$
- $\mathbf{W} \in \mathbb{R}^{784 \times 10}$
- $\mathbf{b} \in \mathbb{R}^{10}$

$$
\phi_{\text{softmax}}(\mathbf{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{\vert \mathbf{z} \vert} e^{z_j}}
$$

$$
\hat{h}(\mathbf{x}; \mathbf{W}, \mathbf{b}) = \phi_{\text{softmax}}(\mathbf{W}^\top \mathbf{x} + \mathbf{b})
$$

$$
\mathcal{H} = \{ h(\mathbf{x}; \mathbf{W}, \mathbf{b}) | \mathbf{W} \in \mathbb{R}^{784 \times 10}, \mathbf{b} \in \mathbb{R}^{10} \}
$$

**Step 2:** use gradient-based optimization methods to obtain $\hat{h}$
Perceptron

\[ \hat{y} = g \left( w_0 + \sum_{i=1}^{m} x_i w_i \right) \]

Linear combination of inputs
Non-linear activation function
Output
Bias

Inputs  Weights  Sum  Non-Linearity  Output

Illustrations by courtesy of [Amini and Soleimany, 2019].
## Activation functions

- **softmax** (identical to sigmoid in binary classification)
- **exponential linear unit (ELU)**
  \[ \phi(z, \alpha)_{\text{ELU}} = \begin{cases} 
  z, & \text{for } z \geq 0 \\
  \alpha(e^z - 1), & \text{for } z < 0 
\end{cases} \]
- **scaled exponential linear unit (SELU)**
  \[ \phi(z, \alpha, \beta) = \beta \cdot \phi(z, \alpha)_{\text{ELU}} \]
- **softsign**
  \[ \phi(z) = \frac{z}{|z| + 1} \]
- **leaky ReLU**
  \[ \phi(z, \alpha) = \begin{cases} 
  z, & \text{for } z \geq 0 \\
  \alpha z, & \text{for } z < 0 
\end{cases} \]
- **parametric ReLU (PReLU, identical to leaky ReLU, but \( \alpha \) is learned)**
  \[ \phi(z; \alpha) = \begin{cases} 
  z, & \text{for } z \geq 0 \\
  \alpha z, & \text{for } z < 0 
\end{cases} \]

<table>
<thead>
<tr>
<th>Activation function</th>
<th>Equation</th>
<th>Example</th>
<th>1D Graph</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unit step (Heaviside)</td>
<td>( \phi(z) = \begin{cases} 0, &amp; z &lt; 0 \ 0.5, &amp; z = 0 \ 1, &amp; z &gt; 0 \end{cases} )</td>
<td>Perceptron variant</td>
<td></td>
</tr>
<tr>
<td>Sign (Signum)</td>
<td>( \phi(z) = \begin{cases} -1, &amp; z &lt; 0 \ 0, &amp; z = 0 \ 1, &amp; z &gt; 0 \end{cases} )</td>
<td>Perceptron variant</td>
<td></td>
</tr>
<tr>
<td>Linear</td>
<td>( \phi(z) = z )</td>
<td>Adaline, linear regression</td>
<td></td>
</tr>
<tr>
<td>Piece-wise linear</td>
<td>( \phi(z) = \begin{cases} 1, &amp; z \leq \frac{1}{2} \ z + \frac{1}{2}, &amp; \frac{1}{2} &lt; z &lt; \frac{1}{2} \ 0, &amp; z \geq \frac{1}{2} \end{cases} )</td>
<td>Support vector machine</td>
<td></td>
</tr>
<tr>
<td>Logistic (sigmoid)</td>
<td>( \phi(z) = \frac{1}{1 + e^{-z}} )</td>
<td>Logistic regression, Multi-layer NN</td>
<td></td>
</tr>
<tr>
<td>Hyperbolic tangent</td>
<td>( \phi(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}} )</td>
<td>Multi-layer Neural Networks</td>
<td></td>
</tr>
<tr>
<td>Rectifier, ReLU</td>
<td>( \phi(z) = \max(0, z) )</td>
<td>Multi-layer Neural Networks</td>
<td></td>
</tr>
<tr>
<td>Rectifier, softplus</td>
<td>( \phi(z) = \ln(1 + e^z) )</td>
<td>Multi-layer Neural Networks</td>
<td></td>
</tr>
</tbody>
</table>

- **only in output layer:**
  - linear activation function (for regression)
  - sigmoid activation function (for binary or multi-label classification)
  - softmax activation function (for binary or multi-class classification)
Single-layer neural network

\[ z_i = w_{0,i}^{(1)} + \sum_{j=1}^{m} x_j w_{j,i}^{(1)} \]

\[ \hat{y}_i = g \left( w_{0,i}^{(2)} + \sum_{j=1}^{d_1} z_j w_{j,i}^{(2)} \right) \]
How deep is deep learning?

Multi-layer neural network

\[ z_{k,i} = w_{0,i}^{(k)} + \sum_{j=1}^{d_{k-1}} g(z_{k-1,j}) w_{j,i}^{(k)} \]

Inputs

\[ x_1 \]
\[ x_2 \]
\[ x_m \]

Hidden

\[ z_{k,1} \]
\[ z_{k,2} \]
\[ z_{k,3} \]
\[ z_{k,d_k} \]

Output

\[ \hat{y}_1 \]
\[ \hat{y}_2 \]
Backpropagation by the chain rule

\[
\frac{\partial J(W)}{\partial w_1} = \frac{\partial J(W)}{\partial \hat{y}} \times \frac{\partial \hat{y}}{\partial z_1} \times \frac{\partial z_1}{\partial w_1}
\]

Repeat this for every weight in the network using gradients from later layers.

Illustrations by courtesy of [Amini and Soleimany, 2019].
Loss Computation shows $y^{(i)}$ is a selector

$$L_{CCE}(\hat{y}^{(i)}, y^{(i)}) = -\sum_{j=1}^{K} y_j^{(i)} \log(\hat{y}_j^{(i)}),$$

$y_j^{(i)} = 1$ only if $x^{(i)}$ belongs to class $j$ and otherwise $y_j^{(i)} = 0$ therefore loss is only realized for class $j$.
Let $c$ be the correct class where the loss is paid

$$\mathcal{L}_{\text{CCE}}(\hat{y}^{(i)}, y^{(i)}) = -\sum_{j=1}^{K} y_{j}^{(i)} \log(\hat{y}_{j}^{(i)}),$$

where

$$\hat{y}_{j}^{(i)} = \frac{\exp(z_{j}^{(i)})}{\sum_{k=1}^{K} \exp(z_{k}^{(i)})},$$

$$\frac{\partial \mathcal{L}_{\text{CCE}}}{\partial z_{j}^{(i)}} = \frac{\partial}{\partial z_{j}^{(i)}} (-z_{c}^{(i)} + \log \sum_{k=1}^{K} \exp(z_{k}^{(i)}))$$

$$\frac{\partial \mathcal{L}_{\text{CCE}}}{\partial z_{j}^{(i)}} = \hat{y}_{j}^{(i)} - 1(j = c)$$
Gradients arise from true class

\[
\frac{\partial L_{\text{CCE}}}{\partial z_j^{(i)}} = \hat{y}_j^{(i)} - 1(j = c)
\]
Updating the weights from gradients

\[ z = W^T x + b \]

\[ z^{(i)}_j = \left( \sum_{k=1}^{N} W_{jk} x_k^{(i)} \right) + b_j \]

\[ \frac{\partial \mathcal{L}_{\text{CCE}}(\theta)}{\partial W_{jk}} = \frac{\partial \mathcal{L}_{\text{CCE}}(\theta)}{\partial z^{(i)}} \frac{\partial z^{(i)}}{\partial W_{jk}} \]

\[ \frac{\partial \mathcal{L}_{\text{CCE}}(\theta)}{\partial W_{jk}} = \frac{\partial \mathcal{L}_{\text{CCE}}(\theta)}{\partial z^{(i)}} x_k \]

\[ W_{jk}^t = W_{jk}^{t-1} - \eta (\hat{y}_j^{(i)} - 1(j = c)) x_k \]

\( \eta \) is the learning rate in this example
Vanishing gradient problem

Higher layers that have small derivatives cause exponential decay of gradients towards the input layers of a network. This is a consequence of the chain rule.
# Categorical cross-entropy loss

def loss_fn(y_pred, y_true):
    return tf.math.reduce_mean(-tf.math.reduce_sum(y_true * tf.math.log(y_pred), axis = 1))

# We can select from a library of gradient management methods here
learning_rate = 0.01
optimizer = tf.keras.optimizers.SGD(learning_rate=learning_rate)

# A training step

def train_step(inputs, labels, opt):
    with tf.GradientTape() as tape:
        predictions = model(inputs)
        pred_loss = loss_fn(predictions, labels)
        dloss_dw, dloss_db = tape.gradient(pred_loss, [W,b])
    opt.apply_gradients(zip([dloss_dw, dloss_db], [W,b]))
    return pred_loss
How complex should our models be? - Overfitting risk

We need to control model complexity to reduce overfitting and enable generalization to unseen examples.

Note hypothesis that deep learning includes an interpolating region that reduces loss past some model capacity.

https://arxiv.org/abs/1812.11118
How complex should our models be? - Overfitting risk

Loss vs. model complexity

https://arxiv.org/abs/1812.11118
Overfitting can also be the result of training too long

Training set ($S_{\text{training}}$):
- set of examples used for learning
- usually 60 - 80% of the data

Validation set ($S_{\text{validation}}$):
- set of examples used to tune the model hyperparameters
- usually 10 - 20% of the data

Test set ($S_{\text{test}}$):
- set of examples used only to assess the performance of fully-trained model
- after assessing test set performance, model must not be tuned further
- usually 10 - 30% of the data
Dropout rate $r$ is probability a node will be eliminated during a given minibatch. We train a thinned network on every minibatch and average the results. Retained weights are scaled by $1/(1 - r)$.

http://jmlr.org/papers/v15/srivastava14a.html

MNIST dropout results (varying architectures)
Weight Regularization

\[ \mathcal{L}_{L1-R}(\theta) = \mathcal{L}_{CCE}(\theta) + \lambda \sum_{i=1}^{N} |\theta_i| \]  
\[ \mathcal{L}_{L2-R}(\theta) = \mathcal{L}_{CCE}(\theta) + \lambda \sum_{i=1}^{N} \theta_i^2 \]  

\[ ||\theta_i||_2 \leq c \]

(L1 Regularization)  
(L2 Regularization)  
(Max Norm Regularization)

We use regularization to control the complexity of models
Graphical Interpretation of L1 and L2 Regularization

$$L_{L1-R}(\theta) = L_{CCE}(\theta) + \lambda \sum_{i=1}^{N} |\theta_i|$$  
(L1 Makes Sparse)

$$L_{L2-R}(\theta) = L_{CCE}(\theta) + \lambda \sum_{i=1}^{N} \theta_i^2$$  
(L2 Minimizes Magnitude)
L1 and L2 Regularization Gradient Changes

\[ \mathcal{L}_{L1-R}(\theta) = \mathcal{L}_{CCE}(\theta) + \lambda \sum_{i=1}^{N} |\theta_i| \]  
\[ \mathcal{L}_{L2-R}(\theta) = \mathcal{L}_{CCE}(\theta) + \lambda \sum_{i=1}^{N} \theta_i^2 \]  

(L1 Regularization)  
(L2 Regularization)

\[ \frac{\partial \mathcal{L}_{L1-R}(\theta)}{\partial \theta_i} = \frac{\partial \mathcal{L}_{CCE}(\theta)}{\partial \theta_i} + \lambda \quad (\theta_i > 0) \]

\[ \frac{\partial \mathcal{L}_{L1-R}(\theta)}{\partial \theta_i} = \frac{\partial \mathcal{L}_{CCE}(\theta)}{\partial \theta_i} - \lambda \quad (\theta_i < 0) \]

\[ \frac{\partial \mathcal{L}_{L2-R}(\theta)}{\partial \theta_i} = \frac{\partial \mathcal{L}_{CCE}(\theta)}{\partial \theta_i} + 2\lambda\theta_i \]
Optimizing objective function

- initialize model parameters $\theta_0, \theta_1, ..., \theta_m$
- repeat until converge, for all $\theta_i$

$$\theta_i^t \leftarrow \theta_i^{t-1} - \eta \frac{\partial}{\partial \theta_i^{t-1}} J(\Theta),$$

where the objective function $J(\Theta)$ is evaluated over all training data $\{(X^{(i)}, y^{(i)})\}_{i=1}^N$.

**Problem Set 1**

**Stochastic Gradient Descent (SGD):** in each step, randomly sample a mini-batch from the training data and update the parameters using gradients calculated from the mini-batch only.
Example convex functions

$c(x) = Mx + b$
$c(x) = e^{c_1(x)}$
$c(x) = c_1(x) + c_2(x)$
$c(x) = |x|$
$c(x) = \max(c_1(x), c_2(x))$

When you compose convex functions the result is not necessarily convex

We are working in the domain of non-convex optimization
Flying into the danger zone

VGG-56 Loss Landscape

VGG-110 Loss Landscape

https://www.cs.umd.edu/~tomg/projects/landscapes/
Gradient Management Methods

- Stochastic gradient descent (update on each training value)
- Mini-batch gradient descent (average gradient over a mini-batch)
- Momentum methods
  - Adam - exponentially decaying average of past gradients and parameter specific learning rates
  - Adadelta - parameters specific learning rates with fixed memory window
Evolution of optimizers

- **Batch gradient descent**: Gradient calculated on entire data set.
- **Stochastic gradient descent**: Gradient calculated for every sample in the dataset per epoch.
- **NAG**: Reduce error by using future step.
- **Momentum**: Throwing the ball downhill - Go faster if going in right direction.
- **Adadelta**: (Decaying average of all past squared gradients)
- **Adagrad**: (Adapts learning rate to the parameters based on previous gradients), but has learning rate shrinking problem.
- **RMSProp**: (Decaying average of all past squared gradients)
- **Adam**: (RMSProp + bias-correction + momentum)
- **Mini-batch gradient descent**: Similar to SGD but on mini batches. Note: This is often referred to as SGD.

**Figure**: Evolution of gradient descent optimization algorithms (image by Desh Raj)
Update equations for different gradient descent optimization algorithms [Ruder, 2016]

<table>
<thead>
<tr>
<th>Method</th>
<th>Update equation</th>
</tr>
</thead>
<tbody>
<tr>
<td>SGD</td>
<td>$g_t = \nabla_{\theta_t} J(\theta_t)$</td>
</tr>
<tr>
<td></td>
<td>$\Delta \theta_t = -\eta \cdot g_t$</td>
</tr>
<tr>
<td></td>
<td>$\theta_t = \theta_t + \Delta \theta_t$</td>
</tr>
<tr>
<td>Momentum</td>
<td>$\Delta \theta_t = -\gamma v_{t-1} - \eta g_t$</td>
</tr>
<tr>
<td>NAG</td>
<td>$\Delta \theta_t = -\gamma v_{t-1} - \eta \nabla_{\theta} J(\theta - \gamma v_{t-1})$</td>
</tr>
<tr>
<td>Adagrad</td>
<td>$\Delta \theta_t = -\frac{\eta}{\sqrt{G_t} + \epsilon} \odot g_t$</td>
</tr>
<tr>
<td>Adadelta</td>
<td>$\Delta \theta_t = -\frac{\eta}{\text{RMS}[\Delta \theta]_{t-1}} g_t$</td>
</tr>
<tr>
<td>RMSprop</td>
<td>$\Delta \theta_t = -\frac{\eta}{\sqrt{E[g^2]}_t + \epsilon} g_t$</td>
</tr>
<tr>
<td>Adam</td>
<td>$\Delta \theta_t = -\frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \hat{m}_t$</td>
</tr>
</tbody>
</table>
Why Deep Residual Networks?

Last two ImageNet Large Scale Visual Recognition Competition (ILSVRC) winners are networks with residual modules:

- ILSVRC 2017: SENet [Hu et al., 2018] is a ensemble of different models, highest scoring model is modified ResNeXt [Xie et al., 2017]
- ILSVRC 2016: ResNet [He et al., 2016]
Evolution of Deep Residual Networks

- **ResNet**: to remedy vanishing gradients, introduce skip connections (bypass layers by adding identity mapping)

![Diagram](image.png)

Figure 2. Residual learning: a building block.
Evolution of Deep Residual Networks

- **Wide Residual Networks**: decrease depth (number of layers), increase width (number of feature maps) in residual layers [Zagoruyko and Komodakis, 2016]
- **ResNeXt**: introduce cardinality (number of aggregated transformations), in addition to depth and width [Xie et al., 2017]

Figure 1. **Left**: A block of ResNet [14]. **Right**: A block of ResNeXt with cardinality = 32, with roughly the same complexity. A layer is shown as (# in channels, filter size, # out channels).
Evolution of Deep Residual Networks

- **DenseNet**: each layer has direct input connections to all previous feature maps [Huang et al., 2016]

**Figure 1**: A 5-layer dense block with a growth rate of $k = 4$. Each layer takes all preceding feature-maps as input.
Less fancy flying required

VGG-56 Loss Landscape

Resnet-56 Loss Landscape

VGG-110 Loss Landscape

Densenet-121

https://www.cs.umd.edu/~tomg/projects/landscapes/

Deep residual learning for image recognition.

Squeeze-and-excitation networks.

Densely connected convolutional networks.
CoRR, abs/1608.06993.

An overview of gradient descent optimization algorithms.
arXiv, 1609.04747.

Aggregated residual transformations for deep neural networks.
In 2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), pages 5987–5995.

Wide residual networks.